PREVENTING EXPLOITS AGAINST MEMORY CORRUPTION VULNERABILITIES

A Thesis
Presented to
The Academic Faculty

by

Chengyu Song

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2016

PREVENTING EXPLOITS AGAINST MEMORY CORRUPTION VULNERABILITIES

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Taesoo Kim, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Professor William R. Harris
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Dr. Weidong Cui
Security and Privacy Research Group
Microsoft Research

Date Approved: 13 July 2016

To my dear wife,

and my parents,

for all the love and support.

ACKNOWLEDGEMENTS

This work owes its existence to the support provided by a large number of people over a time span of several years, and I would like to acknowledge them.

First of all, I would like to thank my terrific advisor Professor Wenke Lee, who has trained me to be an independent researcher. Wenke has always encouraged me to pursue my own interests and provided me the freedom and resources to do so. At the same time, he has also taught me the criteria of good research and the skills of a professional researcher. I am also extreme grateful to my co-advisor Professor Taesoo Kim. Working with him has always been enlightening and inspiring.

I would also like to acknowledge my thesis committee members: Professor Bill Harris, Dr. Weidong Cui, and Professor Mustaque Ahamad for willing to serve on my thesis committee. Their insightful comments and suggestions have helped me make significant improvements to this thesis.

This thesis would not be possible without the help my dear collaborators. Among them, I would like to give special thanks to Dr. Byoungyoung Lee, who has been a good friend and our collaboration has been fruitful. I have also been fortunate to work with the following brilliant researchers: Professor Bill Harris, Dr. Tielei Wang, Dr. Chao Zhang, Paul Royal, Dr. Simon Chung, Dr. Changwoo Min, Dr. Weidong Cui, Mr. Marcus Peinado, Dr. David Melski, Dr. Himanshu Raj, Yeongjin Jang, Kangjie Lu, Insu Yun, Billy Lau, Hyungon Moon, and Professor Alexandra Boldyreva.

I also want to take this opportunity to thank my lab-mates, without them my Ph.D. would not have been the same. I especially would like to acknowledge the following who has been helpful with my research: Monirul Sharif, Junjie Zhang, Long Lu, Xinyu Xing, Brendan Dolan-Gavit, Sangho Lee, Meng Xu, Chenxiong Qian, Ren Ding, and Ming-wei Shih.

Last but not least, I would like to give a special thanks to my family for their continuous

love and support throughout all these years that I have been away from home. The decision to leave my home country and come to the United States to pursue a doctorate was not easy for either of us, but you have always encouraged me to aim high and pursue my dreams. And even though thousands of miles keep us apart, I always carry you in my heart and it gives me strength.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **xi**

LIST OF FIGURES . . . . . **xii**

SUMMARY . . . . . **xiii**

I INTRODUCTION **1**

1.1 Problem Statement and Our Approach 2

1.2 Thesis Contributions 4

1.3 Thesis Outline 5

II BACKGROUND **6**

2.1 Memory Corruption Vulnerabilities 6

2.2 Exploit Techniques 6

2.3 Existing Defense Mechanisms 8

2.3.1 Exploit Prevention 8

2.3.2 Memory Error Detection 12

III PREVENTING CODE INJECTION ATTACKS AGAINST DYNAMIC CODE GENERATOR **15**

3.1 Motivation 15

3.2 Assumptions and Threat Model 17

3.3 Related Work 19

3.3.1 Software-based Fault Isolation 19

3.3.2 Memory Safety 20

3.3.3 Control Flow Integrity 21

3.3.4 Process Sandbox 21

3.3.5 Attacks on JIT engines 22

3.4 Attacking the Code Cache 22

3.4.1 Code Cache Injection Attacks 22

3.4.2 Exploiting Race Conditions 24

3.5 System Design . . . 28
3.5.1 Overview and Challenges . . . 29
3.5.2 Memory Map Synchronization . . . 30
3.5.3 Remote Procedure Call . . . 31
3.5.4 Permission Enforcement . . . 32
3.5.5 Security Analysis . . . 33
3.6 Implementation . . . 33
3.6.1 Shared Infrastructure . . . 33
3.6.2 SDT Specific Handling . . . 35
3.7 Evaluation . . . 38
3.7.1 Setup . . . 38
3.7.2 Effectiveness . . . 38
3.7.3 Micro Benchmark . . . 39
3.7.4 Macro Benchmark . . . 41
3.8 Limitations and Future Work . . . 44
3.8.1 Reliability of Race Condition . . . 45
3.8.2 RPC Stub Generation . . . 45
3.8.3 Performance Tuning . . . 46
3.9 Summary . . . 47

IV PREVENTING KERNEL PRIVILEGE ESCALATION ATTACKS WITH DATA-FLOW INTEGRITY . . . 48

4.1 Motivation . . . 48
4.2 Threat Model and Assumptions . . . 51
4.3 Related work . . . 51
4.3.1 Kernel Integrity . . . 52
4.3.2 Software Fault Isolation . . . 52
4.3.3 Data-flow Integrity . . . 52
4.3.4 Dynamic Taint Analysis . . . 53
4.3.5 Memory Safety . . . 53
4.3.6 Control-flow Integrity . . . 53
4.4 Demonstration Attacks . . . 54

4.4.1 Simple rooting attacks . . . 54
4.4.2 Bypassing CFI with non-control-data attacks . . . 54
4.4.3 Bypassing CFI with control-data attacks . . . 55
4.4.4 Diversity of non-control-data attacks . . . 56
4.5 Technical Approach . . . 56
4.5.1 Inferring Distinguishing Regions . . . 57
4.5.2 Protecting Distinguishing Regions . . . 60
4.6 Formal Model . . . 63
4.6.1 Problem Definition . . . 63
4.6.2 Inferring distinguishing regions . . . 65
4.6.3 Protecting distinguishing regions . . . 68
4.6.4 Protected monitors as refinements . . . 69
4.7 A Prototype for Android . . . 70
4.7.1 Data-flow Isolation . . . 71
4.7.2 MMU Integrity . . . 73
4.7.3 Shadow Objects . . . 74
4.7.4 Kernel Stack Randomization . . . 76
4.8 Evaluation . . . 77
4.8.1 Experimental setup . . . 77
4.8.2 Distinguishing Regions Discovery . . . 78
4.8.3 Security Evaluation . . . 80
4.8.4 Performance Evaluation . . . 80
4.9 Limitations and Future Work . . . 84
4.9.1 Cross-platform . . . 84
4.9.2 Better architecture support . . . 84
4.9.3 Reliability of assumptions . . . 85
4.9.4 Use-after-free . . . 85
4.9.5 DMA protection . . . 85
4.10 Summary . . . 85

V IMPROVE SECURITY AND PERFORMANCE WITH HARDWARE-ASSISTED DATA-FLOW ISOLATION . . . 87

5.1 Motivation . . . 87
5.2 Threat Model and Assumptions . . . 90
5.3 Background and Related Work . . . 90
5.3.1 Data-flow Integrity . . . 90
5.3.2 Tag-based Memory Protection . . . 91
5.3.3 Tag-based Hardware . . . 93
5.3.4 Memory Safety . . . 93
5.4 HDFI Architecture . . . 94
5.4.1 ISA Extension . . . 94
5.4.2 Memory Tagger . . . 95
5.4.3 Optimizations . . . 97
5.4.4 Protecting the Tag Tables . . . 98
5.5 Security Applications . . . 98
5.5.1 Shadow Stack . . . 99
5.5.2 Standard Library Enhancement . . . 100
5.5.3 VTable Pointer Protection . . . 102
5.5.4 Code Pointer Separation . . . 103
5.5.5 Kernel Protection . . . 104
5.5.6 Information Leak . . . 104
5.6 Implementation . . . 105
5.6.1 Hardware . . . 105
5.6.2 Software Support . . . 109
5.6.3 Security Applications . . . 109
5.6.4 Synthesized Attacks . . . 111
5.7 Evaluation . . . 112
5.7.1 Experimental setup . . . 113
5.7.2 Verification . . . 114
5.7.3 Performance Overhead . . . 114
5.7.4 Security Experiments . . . 116
5.7.5 Impact on Existing Security Solutions . . . 117
5.8 Security Analysis . . . 120

5.8.1 Attack Surface 120
5.8.2 Best Practices 121
5.9 Limitations and Future Work 122
5.9.1 DMA Attacks 122
5.9.2 Configurable Tag Table 123
5.9.3 Further Optimizations 123
5.9.4 Dynamic Code Generation 124
5.10 Summary 124
VI CONCLUSION 125
6.1 Summary 125
6.2 Thesis Contributions 126
6.3 Future Work 127
6.4 Closing Remarks 128
REFERENCES 129

LIST OF TABLES

1 RPC Overhead During the Execution of V8 Benchmark. 39

2 Cache Coherency Overhead Under Different Scheduling Strategies. 41

3 SPEC CINT 2006 Results. 41

4 V8 Benchmark Results (IA32). 43

5 V8 Benchmark Slowdown (x64). 43

6 Effectiveness of KENALI against different exploit techniques. 80

7 Compressed kernel binary size increment. 82

8 LMBench results. 83

9 Number of kmem_cache with shadow objects and the number of pages used by shadow objects. 84

10 Comparison between HDFI and other isolation mechanisms. 88

11 Components of HDFI and their complexities in terms of their lines of code. . 104

12 Required efforts in implementing or porting security schemes in terms of lines of code. 109

13 Impact of HDFI on memory read latency (ns), with different optimization techniques. 114

14 Impact of HDFI on memory bandwidth (MB/s), with different optimization techniques. 115

15 Performance overhead of a subset of SPEC CINT 2000 benchmarks. 115

16 The number of total memory read/write access (MB) from both the processor and DFITAGGER. 116

17 Security evaluation of applications utilizing HDFI. 117

18 LMBench results of baseline system and HDFI with kernel stack protection. 119

19 Performance overhead of HDFI-based shadow stack CPS. 120

LIST OF FIGURES

1 A permission switching based W⊕X enforcement. 17
2 Race-condition-based attack using two threads. 17
3 Overview of SDCG's multi-process-based architecture. 29
4 SPEC CINT 2006 Slowdown of Strata. 42
5 V8 Benchmark Slowdown (IA32). 44
6 V8 Benchmark Slowdown (x64). 44
7 Overview of KENALI's approach. 57
8 Shadow address space of KENALI. 70
9 Data fields categorized by their usage. 79
10 Benchmark results from four standard Android benchmarks. 83
11 Design of HDFI. 96
12 Encoding of HDFI's new instructions. 105
13 A simplified diagram of DFITAGGER on a Rocket Chip. 106

SUMMARY

The most common cyber-attack vector is exploit of software vulnerability. Despite much efforts toward building secure software, software systems of even modest complexity still routinely have serious vulnerabilities. More alarmingly, even the trusted computing base (e.g. OS kernel) still contains vulnerabilities that would allow attackers to subvert security mechanisms such as the application sandbox on smartphones. Among all vulnerabilities, memory corruption is one of the most ancient, prevalent, and devastating vulnerabilities. This thesis proposed three projects on mitigating this threat.

There are three popular ways to exploit a memory corruption vulnerability—attacking the code (a.k.a. code injection attack), the control data (a.k.a. control-flow hijacking attack), and the non-control data (a.k.a. data-oriented attack). Theoretically, code injection attack can be prevented with the executable XOR writable policy; but in practice, this policy is undermined by another important technique—dynamic code generation (e.g. JIT engines). In the first project, we first showed that this conflict is actually non-trivial to resolve, then we introduced a new design paradigm to fundamentally solve this problem, by relocating the dynamic code generator to a separate process. In the second project, we focused on preventing data-oriented attacks against operating system kernel. Using privilege escalation attacks as an example, we (1) demonstrated that data-oriented attacks are realistic threats and hard to prevent; (2) discussed two important challenges for preventing such attacks (i.e., completeness and performance); and (3) presented a system that combines program analysis techniques and system designs to solve these challenges.

During these two projects, we found that lacking sufficient hardware support imposes many unnecessary difficulties in building robust and efficient defense mechanisms. In the third project, we proposed HDFI (hardware-assisted data-flow isolation) to overcome this limitation. HDFI is a new fine-grained isolation mechanism that enforces isolation at the machine word granularity, by virtually extending each memory unit with an additional tag

that is defined by data-flow. This capability allows HDFI to enforce a variety of security models such as the Biba Integrity Model and the Bell–LaPadula Model. For demonstration, we developed and ported several security mechanisms to leverage HDFI, including stack protection, standard library enhancement, virtual function table protection, code pointer protection, kernel data protection, and information leak prevention. The evaluation results showed that HDFI is easy to use, imposes low performance overhead, and allows us to create simpler and more secure solutions.

CHAPTER I

INTRODUCTION

Exploits against software vulnerabilities is the most popular attack vector to compromise computer systems. While much effort has been spent on designing, building, and deploying software that is free of defects, software systems of even modest complexity are still routinely deployed with vulnerabilities. More alarmingly, even the trusted computing base (e.g. OS kernel) may contain vulnerabilities that would allow attackers to subvert security mechanisms like the application sandbox on smartphones.

Among all types of vulnerabilities, memory corruption vulnerabilities are one of the worst. For several reasons. First of all, due to the popularity of type unsafe languages (e.g., C/C++), memory corruption vulnerabilities are very common. At the same time, memory corruption vulnerabilities are highly exploitable. As will be discussed in §2, memory corruption vulnerabilities can be exploited in many ways and in most cases they can lead to executing arbitrary logic. As a result, memory corruption vulnerabilities are still one most widely exploited vulnerabilities. According to the latest report from Microsoft [214], memory-corruption-based exploits dominate the remote code execution CVEs.

For severity and popularity of memory-corruption-based attacks, lots of effort has been made to prevent memory-corruption-based exploits. Szekeres et. al [192] did a good summary on existing mitigation mechanisms and their limitations. Generally, existing defense techniques can be put into two categories: memory error detection and exploit mitigation.

Techniques belong to the first category aim at detecting memory errors, the root cause of the vulnerabilities. Since these techniques can stop the attacks from happening in the first place, they have the capability to prevent all memory-corruption-based exploits. However, achieving this is not without cost. Specifically, these techniques tend to have relatively high performance overhead, ranging from 50% to over 100%. Unfortunately, when a trade-off

needs to be made between security and performance, security usually gives the way; hence most techniques are used for offline bug detection or crash analysis.

Exploit mitigation techniques, on the other hand, focus on preventing attackers from performing malicious activities. Specifically, memory corruption vulnerabilities can be exploited in five general ways: code injection attacks, control-flow hijacking attacks, data-oriented attacks, information leak, and data overlapping (§2.2). For each exploit technique, corresponding mitigation mechanisms are developed. For instance, data execution prevention (DEP) is developed to defeat code injection attacks and control-flow integrity (CFI) is proposed to prevent control-flow hijacking attacks. The advantage of exploit prevention techniques is their performance overhead, which is usually less than 10%. For this reason, most deployed mechanisms against memory corruption attacks are exploit mitigation techniques. The downside, however, is that exploit mitigation techniques (at best) can only prevent one type of exploit technique so they can be bypassed.

In short, the state-of-the-art on preventing memory-corruption-based exploit is that, solutions that can provide strong security guarantees are too slow and efficient solutions can only provide limited defense.

1.1 Problem Statement and Our Approach

This thesis aims to *advance the state-of-the-art on defense against memory-corruption-based attacks by building a set of new exploit prevention techniques each of which is capable of blocking one exploit method so the combination of them can approximate the security guarantee of memory error detectors with much less performance overhead.* We choose to build exploit prevention techniques for their low performance overhead and the ease of adoption, i.e., more *practical*. But comparing to existing exploit prevention techniques, our solutions are based on basic security *principles* so they are able to completely block one type of exploit technique and withstand the rapidly evolving arm race from offensive technologies.

The first exploit technique we addressed is code injection attacks. Theoretically, the W$\oplus$X (writable exclusive executable) policy should be able to fully mitigate all code injection attacks. However, this widely deployed technique is not compatible with another important technique:

dynamic code generation. As a result, attackers can leverage dynamic code generators to revive code injection attacks (i.e., code cache injection attack). To fundamentally solve this conflict thus block all code injection attacks, we have developed *Secure Dynamic Code Generation* [184], a novel system design that *fundamentally eliminated the conflict between W⊕X policy and dynamic code generators with small performance overhead.* SDCG achieves these goals through a multi-process-based architecture. Specifically, instead of generating and modifying code in a single process, SDCG relocates the DCG functionality to a second trusted process. The code cache is built upon memory shared between the original process and the trusted process. In the original process, the code cache is mapped as `RX`; but in the trusted process, the same memory is mapped as `WR`. By doing so, the code cache remains read-only under all circumstances in the untrusted process, eliminating the race condition that allows the code cache to be writable to untrusted thread(s). At the same time, the code generator in the trusted process can freely perform code generation, patching, and garbage collection as usual. To enable transparent interaction between the code generator and the generated code, SDCG only need to add a few wrappers that make the code generator callable through remote procedure calls (RPC). Since only functions that modify code cache need to be handled, the effort for adding wrappers is small.

The second exploit technique we addressed is data-oriented attacks. For the popularity of control-flow hijacking attacks, lots of effort of previous work focus on preventing this exploit technique. However, data-oriented attacks can be equally powerful and generic, especially in kernel privilege escalation attacks. To mitigate this threat, we have developed KENALI [182], a system that utilized data-flow integrity (DFI) [34] to *enforce kernel security invariants against memory-corruption-based exploits.* Similar to CFI, DFI guarantees that runtime data-flow cannot deviate from the data-flow graph generated from static analysis. For example, data from a string buffer should never flow to the return address on stack (control-data), or to the `uid` (non-control-data). Utilizing this technique, we can enforce a large spectrum of security invariants in the kernel to defeat different attacks. However, without hardware support, software-based DFI implementation can be very expensive. To overcome this limitation, we developed two novel techniques. Our first technique INFERDISTS can soundly

and automatically infer security-critical memory objects that are vital to kernel privilege escalation attacks. Based on the inference result, our second technique ProtectDists selectively protects these memory objects against memory corruptions attacks with lower performance overhead.

During the development of previous two systems, we found that the isolation mechanisms provided by commodity hardware have become a bottleneck for building strong and efficient solutions against memory corruption attacks. To overcome this limitation, we have developed Hdfi, a novel fine-grained hardware isolation mechanism [183]. Hdfi enforces data isolation at machine word granularity by virtually extending each physical address with an additional tag. Inspired by the idea of data-flow integrity [34], Hdfi defines the tag of a memory unit by the last instruction that writes to this memory location; then at memory read, it allows a program to check if the tag matches what is expected. Utilizing this new hardware feature, we have implemented several defense mechanisms against different memory corruption attacks, including control-flow hijacking prevention (stack protection, standard library enhancement, virtual function table protection, code pointer separation), data-oriented attack prevention (kernel data protection), and information leak prevention (Heartbleed attack). Our development experience shows that Hdfi is *easy to use and usually allows us to create more elegant solutions.* Our security analysis showed that Hdfi-based solutions can *provide better security guarantees than previous implementations.* And our performance showed that, by eliminating data shadowing and context switching, Hdfi can also help *reduce the performance overhead for security mechanisms that requires fine-grained data isolation.*

1.2 Thesis Contributions

In summary, this thesis makes the following technical contributions to the security research community:

- **New Threats Highlighting** With real exploits, we highlight the threats of code cache injection attacks and data-oriented attacks.

- **New Software Design** We present a new software design to resolve the conflict between W⊕X policy and dynamic code generation and to block all code injection attacks.

- **New Program Analysis Technique** The key challenge for building practical defense mechanisms against data-oriented attacks is how to identify data that are vital to those attacks. To solve this challenge, we have develop an automated program analysis technique that can infer data that is critical to kernel privilege escalation attacks.

- **New Isolation Techniques** We develop several techniques to enforce efficient protection over selective memory content and apply them to the Android kernel.

- **New Hardware Design** We develop a new hardware feature to overcome the limitation of the isolation mechanisms provided by commodity hardware. We further demonstrate the benefits of our new hardware feature via developing and evaluating of several defense techniques against memory-corruption-based exploits.

- **Open Source** We will open source all prototype implementations of the techniques presented in this thesis for better real world adoption.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides more details on memory corruption vulnerabilities, including their root causes, common exploit techniques, state-of-the-art defense techniques, and the limitations of existing defense techniques. Chapter 3 discusses the threat of code cache injection attacks and presents the design and evaluation of SDCG. Chapter 4 discusses the threat of data-oriented data (especially in kernel attacks) and presents two new techniques, INFERDISTS and PROTECTDISTS that can fully prevent memory-corruption-based kernel privilege escalation attacks with moderate performance overhead. Chapter 5 discusses the limitation of isolation mechanisms provided by commodity hardware, presents HDFI and how HDFI allows us to build simpler, more secure, and more efficient defense mechanisms against memory-corruption-based exploits. Chapter 6 summarizes the contributions of this thesis and discusses open problems for future work.

CHAPTER II

BACKGROUND

2.1 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities are among the oldest problems of the cyber space. The first publicly recognized worm, the Morris Worm exploited a memory corruption vulnerability to spread itself. At the same time, due to the popularity of type unsafe languages (e.g., assembly, C/C++), memory corruption vulnerabilities are very common. Memory corruption vulnerabilities are also highly exploitable, which usually would allow attackers to read/write arbitrary memory and execute arbitrary code. For these reasons, memory corruption vulnerabilities are one of the most widely exploited vulnerabilities. According to the latest report from Microsoft [214], memory-corruption-based attacks dominate the remote code execution CVEs.

Memory corruption can be caused by three types of memory errors: accessing *uninitialized* memory, accessing *out-of-bound* memory, and accessing *freed* memory. These errors in turn, can be caused by a various of software bugs. For instance, out-of-bound memory access can be caused by lacking of bound check, incorrect bound check (e.g., due to integer overflow/underflow), incorrect allocation (e.g., due to integer overflow), type confusion, etc. Accessing freed memory, a.k.a. use-after-free, can be caused by missing pointer invalidation (nullify) after free, incorrect reference counter update, integer overflow of reference count, etc.

2.2 Exploit Techniques

Once attackers managed to trigger a memory corruption vulnerability, there are three general directions to exploit: leverage it to overwrite memory content, leverage it to read content, or control uninitialized data.

Code injection. The first exploit technique is code injection, where attackers utilized

a memory corruption vulnerability to add new code or modify existing code. Once the injected code gets executed, it would allow attackers to control the system. For its power and ease of construction, code injection attacks used to be the most popular way to launch attacks and shellcode for different tasks is still widely available on the Internet. Even after the introduction of DEP (data execution prevention), code injection is still an important step of modern attacks.

Control-flow hijacking. The second way to exploit a memory corruption vulnerability is to overwrite the control data (a.k.a. code pointers), such as return addresses and function pointers. At the early age of exploit, control-flow hijacking is mostly used to redirect the execution to the injected shellcode. However, after the deployment of DEP, such approach is no longer viable and attackers have moved on to a new technique called *code reuse attacks*. In these attacks, attackers will try to hijack the control-flow to invoke one or more existing code snippets or gadgets. The combination of these code gadgets would allow attacks to perform similar tasks as traditional shellcode or even generic computation [198]. Moreover, researchers have demonstrated that the code gadgets can be of different granularity, which can be as small as a few instructions [176], a basic block [66, 82], or even a whole function [177, 167, 77].

Data-oriented attacks. Besides corrupting code pointers, another way to compromise the integrity of an execution is to tamper with important data of the target program [41, 88, 89]. Data-oriented attacks can be very powerful too. For example, researchers have demonstrated how to compromise browsers [228], attack OS kernel [204, 182], and perform Turing-complete computation [89] with data-oriented attacks.

Information leak. Memory corruption vulnerabilities can also be exploited to perform illegal read and leak important information. In fact, due to the wide deployment of ASLR (address space layout randomization), leaking the address of code is now an inseparable step of most attacks [172, 177]. Moreover, researchers have also demonstrated that information leak can be used to bypass many recently proposed fine-grained randomization-based defense mechanisms [181]. Finally, similar to data-only attacks, attackers can also exploit memory

corruption vulnerabilities to leak generic but important data. For instance, in the Heartbleed attack [47], attackers aimed to leak the private key of a website's certificate.

Data overlapping. Data overlapping is the technique to exploit temporal memory errors (i.e., uninitialized data access and use-after-free). To exploit uninitialized data access, attackers need to control the memory that is not initialized before access; and to exploit use-after-free, attackers need to control the re-allocated memory. By overlapping the attacker-controlled memory with important information, attackers can exploit a temporal memory error to leak such information. For example, uninitialized data read is a popular attack vector to leak kernel information [38]. And by overlapping the attacker-controlled memory with control data (e.g., vtable pointers) or important non-control data(e.g., those used in data-oriented attacks), attackers can also exploit a temporal memory error to launch control-flow hijacking or data-oriented attacks.

2.3 Existing Defense Mechanisms

To defeat memory corruption based attacks, numerous defense mechanisms has been proposed. These mechanisms can be put into two general categories: those aim to eliminate memory errors and those aim to prevent specific exploit techniques. The former approaches could provide better security guarantees as mechanism targeting a specific exploit technique can usually be bypassed. However, due to their much higher performance overhead, in practice, most deployed defense mechanisms are exploit prevention techniques.

2.3.1 Exploit Prevention

Write exclusive Execution. W⊕X is a security policy that requires memory access permissions to be either writable but not executable (e.g., data segments) or be executable but not writable (e.g., code segments). This enforcement can completely mitigate code injection attacks as code becomes not modifiable and data becomes not executable. After the introduction of hardware support for W⊕X enforcement (e.g., `NX`-bit on x86 processors and `XN`-bit on ARM processors), W⊕X has become one of the most widely deployed defense mechanisms. As a result, attackers are forced to leverage more complicated exploit techniques,

such as code reuse attacks. However, the effectiveness of W⊕X can be undermined by another important technique—dynamic code generation (DCG), which usually introduces a code cache that is both writable and executable. In §3, we will show how this DCG can be utilized to revive code injection attacks and how to fundamentally solve this weakness of W⊕X.

Randomization. Randomization is a generic way to detect/prevent "foreign", attacker introduced data. Currently, the most widely deployed randomization-based mechanism is ASLR [156], which randomized the memory layout hence pointers of a process. With ASLR, if attackers try to overwrite/overlap a security critical pointer (e.g., return address) with their crafted value, the pointer is likely to be invalid. Besides the coarse-grained layout randomization, other types of randomization have also been proposed. ASLP [105] randomizes the layout of functions, binary stirring [210] randomizes the layout of basic blocks, ICR [152] randomizes instruction layout within a basic block, ILR [87] randomizes the layout of all instructions, DSLR [118] randomizes the layout of data structures, and ISR [161, 20] randomizes the encoding of instructions.

Randomization-based defense mechanisms have two weaknesses: lack of entropy and information leak. Without enough entropy, randomization-based defense mechanism can be defeated by spray-based attacks [61, 179] or brute-force-based attacks [177, 25, 76]. And if the randomized information can be leaked (e.g., via memory corruption or side-channel attacks [91, 81]), all randomization-based approaches can be bypassed [181].

ret2usr Prevention. `ret2usr` is a special type of control-flow hijacking attacks that redirect the control-flow to user mode code while the processor is in privileged mode. As a result, attackers can perform operations that should only be allowed by the OS kernel. KERNEXEC [155] uses the `CS` segment register to restrict the code range of in kernel mode. kGuard [104] uses a lightweight range check to prevent indirect control transfer to user space. SMEP (Supervisor Mode Execution Protection) and PXN (Privileged eXecution Not) are hardware features to prevent such attacks.

Control-flow Integrity. For the popularity of control-flow hijacking attacks, many techniques have been proposed to prevent illegal control transfer. Stack canary [53] is

the first solution to prevent stack buffer overflow from tampering the return addresses and is supported by most compilers. However, it cannot prevent direct (not sequential) overwrite and may be subject to brute-force-based attacks [25]. Another popular approach is shadow stack [201, 45, 60] which stores return addresses to a separate stack. An alternative approach is called safe stack [108], which relocates unsafe allocations to a separate stack. The challenge for these approaches is how to protect the safe stack from arbitrary memory writes [76]. To solve this challenge, hardware-protected shadow stack has also been proposed [218, 114, 150, 63, 95]. Finally, solutions have also been proposed to detect return-oriented programming (ROP) [153, 44].

After securing the backward-edge control-flow, the next step is to secure the forward-edge control transfer. To solve this challenge, Abadi et al. [1] introduced the term control-flow integrity (CFI) which enforces that the runtime control-flow transfer should not deviate from the control-flow graph (CFG) constructed from static analysis. Since then, many following up work have been proposed to improve different aspects of this approach. HyperSafe [209] extends CFI to hypervisors; KCoFI [56] extends CFI to monolithic OS kernel; CCFIR [233] and binCFI [234] try to enforce coarse-grained CFI (due to loss of information) for COST binaries; MCFI [143] tries to bring modular support; RockJIT [144] and JITScope [231] extends CFI to dynamically generated code; π-CFI [145] tries to improve the precision of CFG by dynamically enable transfer targets based on current execution trace; and various hardware-based CFI solutions aim to improve the efficiency of the enforcement [55, 103, 64, 46, 95].

One thing to notice is that, a CFI solution must cover both forward-edge and backward-edge, otherwise attacks are still feasible [51]. Despite problems of individual CFI implementations [33, 66, 82], the more fundamental problems of CFI are (1) due to the theoretical limitation of static analysis, control-flow hijacking attacks are still feasible without deviating from the CFG [77] and (2) data-oriented attacks.

Virtual Function Call. Virtual function is an important feature of the C++ language and consists a majority of indirect calls in C++ programs. This mechanism has been abused by attackers to launch control-flow hijacking attacks by injecting/modifying the virtual function table (vtable) pointers. To defeat such attacks, several solutions have

been proposed. SafeDispatch [97] is similar to CFI and restricts the target function to be within the set generated from class hierarchy analysis. FCFI [197] takes a similar approach but only validates the vtable pointers and supports incremental build. VTint [232] uses binary rewriting to prevent vtable injection attacks, but cannot prevent vtable reuse attacks. VTable interleaving [27] tries to reduce the cost of validation checks with a new vtable layout. VTrust [230] uses a multi-layer protection scheme to defeat all vtable-related attacks with low performance overhead.

Pointer Encryption. Because many exploit techniques involve injecting/overwriting pointers, another direction to defeat such attacks is to protect the integrity of pointers. PointGuard [54] encrypts all pointers in memory and only decrypts them when pointers are loading into the registers. The goal of the encryption is to prevent attackers from generating pointers controlled by them. Unfortunately, as the encryption scheme used by PointGuard is too simple—XOR with one single key, this key can be easily recovered [189]. CCFI [124] enforces CFI by encrypting all code pointers (function pointers, vtable pointers, and exception handlers) and return addresses with AES (hardware supported AES-NI instructions) to create a MAC (message authentication code). If attackers try to inject their own pointers, then the decrypted pointers will not pass the MAC check. ASLR-Guard [120] encrypts all pointers that can be used to infer the address of code so as to prevent breaking ASLR with information leak. This encryption also prevents attackers from modifying protected pointers to transfer the control-flow to arbitrary destinations. The drawback of encryption is replay attacks, i.e., with information leak, attackers can still launch attacks by reusing valid pointers.

Isolation. Isolation is a generic approach to prevent attackers from tampering important data thus can be used to prevent data-oriented attacks. An isolation-based protection mechanism faces two challenges. First, efficient hardware-based isolation supports are removed in 64-bit mode—segments on x86 processors and access domain on ARM processors are both gone. As a result, security solutions must choose between efficiency and security. In §5, we will discuss more about this problem and present a new hardware feature to solve this problem. The second challenge is how to identify what data is critical and should be

protected/isolated. Many existing solutions identify such data based on heuristics or existing attacks. As a result, they cannot provide enough security guarantees thus can be bypassed. In §4, we will show a new approach to systematically discover critical data in the kernel space.

Information Flow Tracking. Compare to control-flow hijacking, preventing memory-corruption-based information leak prevention is relatively less popular topic. Among existing solutions, the most generic one is dynamic information flow tracking (DIFT) [224, 73].

Execute-only Memory. To prevention information leak from breaking fine-grained randomization techniques, researchers have also proposed execute-only (i.e., neither readable nor writable) memory [17, 65], which is also supported by latest Intel processors through "memory protection keys."

2.3.2 Memory Error Detection

Uninitialized Memory. Most compilers provide the `-Wuninitialized` option to detect accessing uninitialized stack variables, but the scope is limited to the current function. Besides compile-time detection, many runtime memory error detection tools also support detecting uninitialized memory access. Valgrind's memcheck [174] and Dr. Memory [29] use dynamic binary instrumentation to monitor all memory access of the target program. The performance overhead of these tools can be above 10x. kmemcheck [146] uses shadow memory to track the state of allocated memory, but only support heap objects allocated from the standard heap allocator. MemorySanitizer [188] relies on compile-time instrumentation and shadow memory to detect uninitialized data use at run-time. Its performance is much better than dynamic instrumentation based tools; however, it still imposes a 3x-4x performance overhead. Usher [222] proposed value-flow analysis to reduce the number of tracked allocations and reduced the performance overhead of MemorySanitizer to 2x-2.5x.

Spatial Error. There are two general directions to detect spatial memory errors: object-based (i.e., can the pointer used to access the target memory) and pointer-based (i.e., is the target address out-of-bound). Memcheck [174], Dr. Memory [29], kmemcheck [146], and AddressSanitizer [171] are representative shadow-memory based error detectors. They use

shadow memory to mark which memory address can be accessed and which cannot. Yong et al. [225], write integrity test (WIT) [4] and data-flow integrity (DFI) [34] uses static point-to analysis to decide whether the target memory can be written by the given pointer. Compared to pointer-based approaches, object-based approaches all suffer from imprecision thus may still allow some attacks to happen.

J&K [100], CRED [165], Baggy Bounds Checking (BBC) [5], and PAriCheck [227] try to detect out-of-bound pointers at the time of pointer creation, i.e., whether the result of a pointer arithmetic would be out-of-bound. The drawback of this direction is that, because bound information is created at memory allocation, they cannot detect memory corruption within the allocated object.

CCured [137] and Cyclone [98] retrofit the C language to use "fat-pointers" (i.e., pointer with additional boundary information) for memory access and to perform boundary check for unsafe memory dereference. Unfortunately, because they both require source code level changes and are not binary compatible with existing code, they have not been adopted in practice. MSCC [219] uses source code transformation to avoid source code change and more compact metadata to improve efficiency. SoftBound [135] uses compiler-based instrumentation to avoid source code change and split metadata to solve memory the layout compatibility problem.

The common problem of all spatial memory error detector, especially for those based on boundary check, is high performance overhead, which is usually more than 100%. For this reason, researchers have also proposed special hardware to improve the performance and compatibility (i.e., without re-compilation) [69, 133, 134, 212]. However, even with hardware support, memory safety enforcement can still impose 29% slowdown on SPEC CINT 2006 benchmarks [134].

Use-after-free. DieHard [23] and DieHarder [147] use special memory allocators to approximate infinite memory so as to avoid memory reuse. With state tracking, Memcheck [174], Dr. Memory [29], kmemcheck [146], and AddressSanitizer [171] can also detect temporal memory errors, as long as that memory has not be reallocated. CETS [136] uses version tracking to detect whether a pointer is a dangling pointer (i.e., points to a freed object).

DangNull [113] and FreeSentry [226] uses pointer nullification to eliminate dangling pointers when the referenced memory is freed. CCured [137], Cyclone [98], and MemGC [129] solve problem by relying on garbage collection.

CHAPTER III

PREVENTING CODE INJECTION ATTACKS AGAINST DYNAMIC CODE GENERATOR

3.1 Motivation

Exploits against memory corruption vulnerabilities remain one of the most severe threats to cyber security. To mitigate this threat, many techniques have been proposed, including data execution prevention (DEP) [6] and address space layout randomization (ASLR) [156], both of which have been widely deployed and are effective. DEP is a subset of the more general security policy W⊕X, which enforces that memory should either be writable but not executable (e.g., data segments), or be executable but read-only (e.g., code segments). This enforcement can completely mitigate traditional exploits that corrupt existing code or inject malicious shellcode into data segments. Consequently, attackers have to leverage more complicated exploit techniques, such as return-to-libc [177] and return-oriented-programming (ROP) [176]. Moreover, W⊕X memory has become the foundation of many other protection techniques, such as control flow integrity (CFI) [1, 234, 233].

However, the effectiveness of W⊕X can be undermined by another important compilation technique—dynamic code generation (DCG). With the ability to generate and execute native machine code at runtime, DCG is widely used in just-in-time (JIT) compilers [15] and dynamic binary translators (DBT) [169, 121] to improve performance, portability, and security. For example, JIT compilers for dynamic languages (e.g., JavaScript and ActionScript) can leverage platform information and runtime execution profile to generate faster native code. DBTs can leverage DCG to provide dynamic analysis capability [121], cross-platform or cross-architecture portability [168, 22], bug diagnostics [139, 163], and enhanced security [20, 142, 43, 90, 87].

A fundamental challenge posed by DCG is that the code cache, in which the dynamically generated code is stored, needs to be both writable (for code emitting, code patching, and

garbage collection) and executable. This violates the W⊕X policy and enables a new attack vector. In fact, security researcher has already demonstrated the feasibility of exploiting this weakness through a real world exploit that delivers shellcode into the writable code cache and successfully compromises the Chrome web browser [160].

Solving this problem seems trivial. A straightforward idea, which has been adopted in browsers like mobile Safari, is demonstrated in Figure 1. This technique keeps the code cache as read-only and executable (`RX`) when the generated code is executing; switches to writable but not executable (`WR`) when the code cache needs to be modified ($t1$); and switches back to `RX` when the write operation finishes ($t2$). As a result, the code cache will remain read-only when the generated code is executing; and the attack demonstrated in [160] can be mitigated.

Unfortunately, in addition to performance overhead, this simple mechanism does not work well with multi-threaded programs. First, if the code generator uses a shared code cache for all threads (e.g., PIN [121]), then the code cache cannot be switched to `WR`, because other concurrently running threads require the executable permission. Second, even if the code generator uses a dedicated code cache for each thread (e.g., JS engines), the protection is still flawed and is subject to *race condition* attacks [140], as shown in Figure 2. More specifically, memory access permissions are applied to the whole process and are shared among all threads. When one thread turns on the writable permission for its code cache (e.g., for code emitting), the code cache also becomes writable to all other threads. Once the write permission is set, another concurrently running thread can (maliciously) overwrite the first thread's code cache to launch attacks. This is similar to the classic time-of-check-to-time-of-use (TOCTTOU) problem [126], where the resource to be accessed is modified between the check and the use by exploiting race conditions.

In this chapter, we demonstrate the feasibility of such race-condition-based code cache injection attacks, through a proof-of-concept exploit against modern browsers that support the *Web Worker* [205] specification. Rather than relying on a permanently writable code cache [160], our attack leverages race conditions and can bypass permission-switching-based W⊕X enforcement (Figure 1). In this attack, the malicious JS code utilizes web workers to

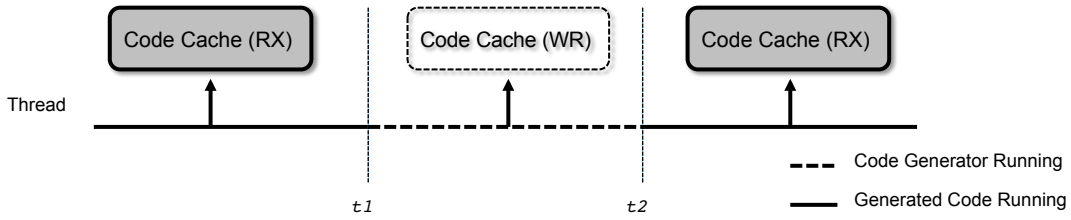

Figure 1: A permission switching based W⊕X enforcement.

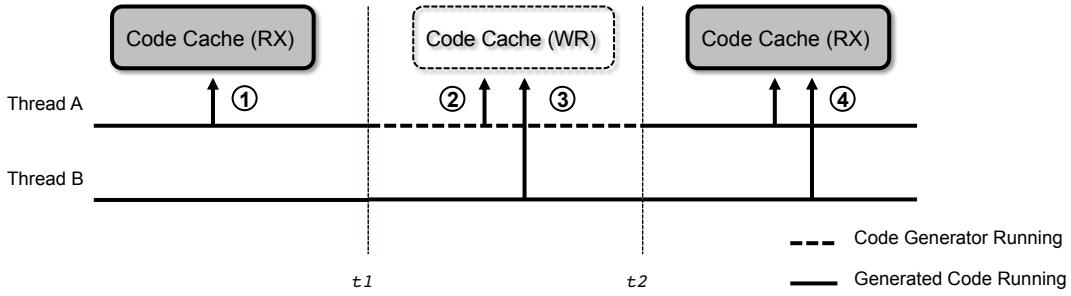

Figure 2: Race-condition-based attack using two threads.

create a multi-threaded environment. After forcing a worker thread into the compilation state, the main JS thread can exploit vulnerabilities of the browser to inject shellcode into the worker thread's code cache.

To fundamentally prevent such attacks, we propose secure dynamic code generation (SDCG), a new architecture that (1) enables dynamic code generation to comply with the W⊕X policy; (2) eliminates the described race condition; (3) can be easily adopted; and (4) introduces less performance overhead compared to alternative solutions.

3.2 Assumptions and Threat Model

SDCG focuses on preventing remote attackers from leveraging memory corruption vulnerabilities to overwrite the code cache to achieve arbitrary code execution. Hence, we focus on two classic attack scenarios discussed as follows. In both scenarios, we assume the code generator itself is trusted and does not have security vulnerabilities.

- *Foreign Attacks.* In this scenario, the code generator is a component of a program (e.g., a web browser). The program is benign, but components other than the code generator are assumed to be vulnerable when handling input or contents provided by an attacker (e.g., a malicious web page). Attackers can then exploit the vulnerable components to attack the code cache.

- *Jailbreak Attacks.* In this scenario, the code generator is used to sandbox or monitor an untrusted program, and attacks are launched within the code cache. This could happen under two circumstances. First, the sandboxed program itself is malicious. Second, the program is benign, but the dynamically generated code has vulnerabilities that can be exploited by attackers to jailbreak.

Without loss of generality, we assume that the following mitigation mechanisms for both general and JIT-based exploits have been deployed on the target system.

- *Address Space Layout Randomization.* We assume that the target system has at least deployed the basic ASLR and all predictable memory mappings have been eliminated.

- *JIT Spray Mitigation.* For JIT engines, we assume that they have implemented a full-suite of JIT spray mitigation mechanisms, including but not limited to random NOP insertion, constant splitting, and those proposed in [213, 9].

- *Guard Pages.* We assume the target system creates guard pages (i.e., pages without access permission) to wrap each pool of the code cache, as seen in the Google V8 JS engine. These guard pages can prevent buffer overflows, both overflows out of the code cache and overflows into the code cache.

- *Page Permissions.* We assume that the underlying hardware has support for mapping memory as non-executable (NX) and that writable data memory like the stack and normal heap are set to be non-executable. Furthermore, we assume that all statically generated code has been set to non-writable to prevent overwriting. However, almost all JIT compilers map the code cache as both writable and executable.

The target system can further deploy the following advanced mitigation mechanisms for the purpose of sandboxing and monitoring:

- *Fine-grained Randomization.* The target system can enforce fine-grained randomization by permuting the location of functions [105], basic blocks [210], and each instruction [152]; and even randomizing the instruction set [20].

- *Control Flow Hijacking Mitigation.* The target system can deploy control flow hijacking mitigation mechanisms, including (but not limited to): control flow integrity enforcement, either coarse-grained [233, 234] or fine-grained [1, 143]; return-oriented programming detection [153, 44]; and dynamic taint analysis based hijacking detection [142].

To allow overwriting of the code cache, we assume there is at least one vulnerability that allows attackers to write to an attacker-specified address with attacker-provided contents. We believe this is a realistic assumption, because many types of vulnerabilities can be exploited to achieve this goal, such as format string [141], heap overflow [50], use-after-free [48], integer overflow [49], etc. For example, the attack described in [160] obtained this capability by exploiting an integer overflow vulnerability (CVE-2013-6632); in [42], the author described how five use-after-free vulnerabilities (CVE-2013-0640, CVE-2013-0634, CVE-2013-3163, CVE-2013-1690, CVE-2013-1493) can be exploited to perform arbitrary memory writes. It is worth noting that in many attack scenarios, the ability to do arbitrary memory write can easily lead to arbitrary memory read and information disclosure abilities.

3.3 Related Work

In this section, we discuss the techniques that could be used to protect a code cache from being maliciously modified and explain their limitations. We also discuss other forms of attacks against the JIT engines and their countermeasures.

3.3.1 Software-based Fault Isolation

Software-based fault isolation (SFI) [207] can be used to confine a program's ability to access memory resources. On 32-bit x86 platforms, SFI implementations usually leverage segment registers [79, 223] to confine memory accesses for the benefit of low runtime overhead. On other platforms without segment support (e.g., x86-64, ARM), SFI implementations use either address masking [170, 144] or access control lists (ACL) [35], introducing higher runtime overhead.

Once memory accesses, especially write accesses are confined, SFI can prevent untrusted

code from overwriting security sensitive data, such as the code cache. SDCG differs from SFI in several respects. First, SFI's overhead comes from the execution of the extra inline checks while SDCG's overhead comes from remote procedure calls and cache synchronization on multi-core systems. Therefore, if execution stays mostly within the code cache, SDCG will introduce less overhead than SFI. On the other hand, if execution needs to be frequently switched between the code generator and the generated code, then SFI could be faster. Since most modern code generators try to make the execution stay as long as possible in the code cache, our approach is more suitable in most cases.

Second, to reduce the overhead of address masking, many SFI solutions [170, 144] use ILP32 (32-bit integer, long, pointer) primitive data types, limiting data access to 4GB of space, even on a 64-bit platform. SDCG does not have this limitation.

It is worth noting that some efforts have been made to apply SFI to JIT engines [9, 144]. Despite relatively higher overhead, the threat model of these approaches usually did not consider scenarios where the JIT compiler is only a component of a larger software, such as a web browser. Since most web browser vulnerabilities are found outside the JIT engines [58], to apply such techniques one would have to apply SFI to other browser components as well. This could result in even higher performance overhead. From this perspective, we argue that our solution is more realistic in practice.

3.3.2 Memory Safety

Attacking code caches (at randomized locations) relies on the ability to write to a memory location specified by attackers. Therefore, such attacks could be defeated by memory safety enforcement, which prevents all unexpected memory reads and writes. However, many programs are written in low-level languages like C/C++, and are prone to memory corruption bugs, leading to the majority of security vulnerabilities for these languages. Unfortunately, existing memory safety solutions [175, 71, 14, 154, 219, 136, 135] for C/C++ programs tend to have much higher performance overhead than SFI or other solutions, prohibiting their adoptions. For example, the combination of Softbound [135] and CETS [136] provides a strong spatial and temporal memory safety guarantee, but they were reported to have 116%

average overhead on SPEC CPU 2000 benchmark. Compared with this direction of research, even though SDCG provides less security guarantees, it is still valuable because it fully blocks a powerful attack vector with much lower runtime overhead.

3.3.3 Control Flow Integrity

Control flow hijacking is a key step in many real world attacks. As DEP becomes ubiquitous, more and more attacks rely on return-to-libc [177] or ROP [176] to hijack control flow. Many solutions [1, 234, 233, 143, 144, 231] have been proposed to enforce control flow integrity (CFI) policy. With CFI policy, a program's control flow cannot be hijacked to unexpected locations. CFI could protect the code cache in some way, e.g., attackers cannot overwrite the code cache by jumping to arbitrary addresses of the code generator.

However, attackers can still utilize arbitrary memory write vulnerabilities to overwrite the code cache without breaking CFI. Once the code cache is overwritten, injected code could be invoked through normal function invocations without breaking the static CFI policy.

More importantly, when extending CFI to dynamically generated code [144, 231], without proper write protection the embedded enforcement checks can also be removed once attackers can overwrite the code. From this perspective, SDCG is complementary to CFI because it guarantees one basic assumption of CFI: code integrity protection.

3.3.4 Process Sandbox

A delegation-based sandbox architecture, a.k.a. the broker model [80], has been widely adopted by the industry and used in Google Chrome [84], Windows 8 [128], Adobe Reader [2], etc. In this architecture, the sandboxed process drops most of its privileges and delegates all security sensitive operations to the broker process. The broker process then checks whether the request complies with the security policy. SDCG is based on the same architecture. Using this architecture, we (1) delegate all the operations that will modify the code cache (e.g., code installation, patching, and deletion) to the translator process; and (2) make sure the W⊕X policy is mandatory.

3.3.5 Attacks on JIT engines

Attackers have targeted the code cache for its writable and executable properties. Currently, the most popular exploit technique is JIT spray [179], an extension to classic heap spray attacks [61]. Heap spray is used to bypass ASLR without guessing the address of injected shellcode. This technique becomes infeasible after DEP is deployed because the heap is no longer executable. To bypass this, attackers turned to JIT engines. The JIT spray attack abuses the JIT engine to emit chunks of predictable code and then hijacks control flow toward the entry or middle of one of these code chunks. DEP or W⊕X is thus bypassed because these code chunks reside in the executable code cache. Most JIT engines have since deployed different mitigation techniques to make the layout of the code cache unpredictable, e.g., random NOP insertion, constant splitting, etc. Researchers have also proposed more robust techniques [213, 9] to prevent such attacks.

Rather than abusing JIT engines to create expected code, attackers can also abuse the writable property of the code cache and directly overwrite generated code [160]. In the next section, we first extend this attack to show that even with a permission switching based W⊕X enforcement, attackers can still leverage race conditions to bypass such enforcement.

3.4 Attacking the Code Cache

This section describes in detail the code cache injection threat SDCG aims to address. First, we show how the code cache can be attacked to bypass state-of-the-art exploit mitigation techniques. Then we demonstrate how a naive W⊕X enforcement can be bypassed by exploiting race conditions.

3.4.1 Code Cache Injection Attacks

Software Dynamic Translator. For ease of discussion, we use the term software dynamic translator (SDT) to represent software that leverages dynamic code generation to translate code in one format to another format. Before describing the attacks, we first give a brief introduction on SDT. A core task of all SDTs is to maintain a mapping between untranslated code and translated code. Whenever a SDT encounters a new execution unit (depending on

the SDT, the execution unit could be a basic block, a function, or a larger chunk of code), it first checks whether the execution unit has been previously translated. If so, it begins executing the translated code residing in the code cache; otherwise, it translates this new execution unit and installs the translated code into the code cache.

Exploit Primitives. In this subsection, we describe how the code cache with full `WRX` permission can be overwritten. This is done in two steps. First, we need to bypass ASLR and find out where the code cache is located. Second, we need to write to the identified location.

Bypassing ASLR. The effectiveness of ASLR or any randomization based mitigation mechanism relies on two assumptions: (i) the entropy is large enough to prevent brute-force attacks; and (ii) the adversary cannot learn the random value (e.g., module base, instruction set). Unfortunately, these two assumptions rarely hold in practice. First, on 32-bit platforms, user space programs only have 8 bits of entropy for heap memory, which is subject to brute-force guessing [177] and spray attacks [61]. Second, with widely available information disclosure vulnerabilities, attackers can easily recover the random value [172, 164]. In fact, researchers have demonstrated that even with a single restricted information disclosure vulnerability, it is possible to traverse a large portion of memory content [181].

When attacking a code cache, we can either launch a JIT spray attack to prepare a large number of `WRX` pages on platforms with low entropy, or leverage an information disclosure vulnerability to pinpoint the location of the code cache. Note that as one only needs to know the location of the code cache, most fine-grained randomizations that try to further randomize the contents of memory are ineffective against this attack. Since the content of code cache will be overwritten in the next step (described below), none of the JIT spray mitigation mechanisms can provide effective protection against this attack.

Writing to the Code Cache. The next step is to inject shellcode to the code cache. In most cases, the code cache will not be adjacent to other writable heap memory (due to ASLR), and may also be surrounded by guard pages. For these reasons, we cannot directly exploit a buffer overflow vulnerability to overwrite the code cache. However, as our assumption section (§3.2 suggests, besides logic errors that directly allow one to write to

anywhere in memory, several kinds of memory corruption vulnerabilities can also provide arbitrary memory write ability. In the following example, an integer overflow vulnerability is exploited to acquire this capability.

An In-the-Wild Attack. We have observed one disclosed attack [160] that leveraged the code cache to achieve reliable arbitrary code execution. This attack targeted the mobile Chrome browser. By exploiting an integer overflow vulnerability, the attack first gained reliable arbitrary memory read and write capabilities. Using these two capabilities, the attack subsequently bypassed ASLR and located the permanently writable and executable code cache. Then it injected shellcode into the code cache to overwrite an emitted JS function. Finally, by invoking the corrupted function, it turned control flow to the shellcode.

Security Implication. In practice, we have only observed this single attack that injects code into the code cache. We believe this is mainly due to the convenience of a popular ROP attack pattern, which works as: (i) preparing traditional shellcode in memory; (ii) exploiting vulnerabilities to launch an ROP attack; (iii) using the ROP gadgets to turn on the execution permission of memory where the traditional shellcode resides; and (iv) jumping to the traditional shellcode to finish the intended malicious tasks. However, once advanced control flow hijacking prevention mechanisms such as fine-grained CFI are deployed, this attack pattern will be much more difficult to launch.

On the other hand, code cache injection attack can easily bypass most of the existing exploit mitigation mechanisms. First, all control flow hijacking detection/prevention mechanisms such as CFI and ROP detection rely on the assumption that the code cannot be modified. When this assumption is broken, these mitigation mechanisms are no longer effective. Second, any inline reference monitor based security solution is not effective because the injected code is not monitored.

3.4.2 Exploiting Race Conditions

A *naive* defense against the code cache injection attack is to enforce W⊕X by manipulating page permissions (Figure 1). More specifically, when the code cache is about to be modified (e.g., for new code generation or runtime garbage collection), the code generator turns on the

write permission and turns off the execution permission ($t1$). When the code cache is about to be executed, the generator turns off the write permission and turns on the execution permission ($t2$).

This solution prohibits the code cache to be both writable and executable at the same time. If the target program is single-threaded, this approach can prevent code cache injection attacks, as the code cache is only writable when the SDT is executing and we assume that the SDT itself is trusted and not vulnerable, attackers cannot hijack or interrupt the SDT to overwrite the code cache. However, as illustrated in Figure 2, in a more general multi-threaded programming environment, even if the SDT is trusted, the code cache can still be overwritten by other insecure threads when the the code cache is set to be writable for one thread.

In this subsection, we use a concrete attack to demonstrate the feasibility of such attacks, i.e., with naive W⊕X enforcement, it is still possible to overwrite the code cache with the same exploit primitives described above.

Multi-threaded Programming in SDT. To launch the race-condition-based attack, we need two more programming primitives. First, we need the ability to write multi-threaded programs. Note that some SDTs such as Adobe Flash Player also allows "multi-threaded" programming, but each "thread" is actually implemented as a standalone OS process. For these SDTs, since the code cache is only writable to the corresponding "thread", our proposed exploit technique would not work. Second, since the attack window is generally small, we need the ability to coordinate threads before launching the attack.

- *Thread Primitives.* A majority of SDTs have multi-threaded programming support. JavaScript (JS) used to be single-threaded and event-driven. With the new HTML5 specification, JS also supports multi-threaded programming through the `WebWorker` specification [205]. There are two types of `WebWorker`: *dedicated* worker and *shared* worker. In V8, the dedicated worker is implemented as a thread within the same process; a shared worker is implemented as a thread in a separate process. Since we want to attack one JS thread's code cache with another JS thread, we leverage the

dedicated worker. Note that although each worker thread has its own code cache, it is still possible to launch the attack, because memory access permissions are shared by all threads in the same process.

- *Synchronization Primitives.* To exploit the race condition, two attacker-controlled threads need to synchronize their operations so that the overwrite can happen within the exact time window when the code cache is writable. Since synchronization is an essential part of multi-threaded programming, almost all SDTs support thread synchronization. In JS, thread synchronization uses the `postMessage` function.

A Proof-of-Concept Attack. Based on the vulnerability disclosed in the previous real-world exploit, we built a proof-of-concept race-condition-based attack on the Chrome browser. Since the disclosed attack [160] already demonstrated how ASLR can be bypassed and how arbitrary memory write capability can be acquired, our attack focuses on how race conditions can be exploited to bypass naive W⊕X enforcement. The high level workflow of our attack is as follows:

1. *Create a Worker.* The main JS thread creates a web worker, and thus a worker thread is created.

2. *Initialize the Worker.* The worker thread initializes its environment, making sure the code cache is created. It then sends a message to the main thread through `postMessage` that it is ready.

3. *Locate the Worker's Code Cache.* Upon receiving the worker's message, the main JS thread locates the worker thread's code cache, e.g., by exploiting an information disclosure vulnerability. In the Chrome V8 engine, attackers can locate the code cache using the previously disclosed exploit. Instead of following the pointers for the current thread, attackers should go through the thread list the JS engine maintains and follow pointers for the worker thread. Then, the main thread informs the worker that it is ready.

4. *Make the Code Cache Writable.* Upon receiving the main thread's message, the worker thread begins to execute another piece of code, forcing the SDT to update its code cache. In V8, the worker can execute a function that is large enough to force the SDT to create a new `MemoryChunk` for the code fragment and set it to be writable (for a short time).

5. *Monitor and Overwrite the Code Cache.* At the same time, the main thread monitors the status of the code cache and tries to overwrite it once its status is updated. In V8, the main thread can keep polling the head of the `MemoryChunk` linked list to identify the creation of a new code fragment. Once a new code fragment is created, the main thread can then monitor its content. Once the first few bytes (e.g., the function prologue) are updated, the main thread can try to overwrite the code cache to inject shellcode. After overwriting, the main thread informs the worker it has finished.

6. *Execute the Shellcode.* Upon receiving the main thread's new message, the worker calls the function whose content has already been overwritten. In this way, the injected shellcode is executed.

It is worth noting that the roles of the main thread and the worker thread cannot be swapped in practice, because worker threads do not have access to the document object model (DOM). Since many vulnerabilities exist within the rendering engine rather than the JS engine, this means only the main thread (which has the access to the DOM) can exploit those vulnerabilities.

Reliability of the Race Condition. One important question for any race-condition-based attack is its reliability. The first factor that can affect the reliability of our attack is synchronization, i.e., the synchronization primitive should be fast enough so that the two threads can carry out the attack within the relatively small attack window. To measure the speed of the synchronization between the worker and the main thread, we ran another simple experiment:

1. The main thread creates a worker thread;

2. The worker thread gets a timestamp and sends it to the main thread;

3. Upon receiving the message, the main thread sends an echo to the worker;

4. Upon receiving the message, the worker thread sends back an echo;

5. The main thread and the worker repeatedly send echoes to each other 1,000 times.

6. The main thread obtains another timestamp and computes the time difference.

The result shows that the average synchronization delay is around 23 μs. The average attack window ($t2 - t1$ in Figure 2) of our fine-grained naive W$\oplus$X protection is about 43 μs. Thus, in theory, the `postMessage` method is sufficiently fast to launch the race condition attack.

The second and more important factor that can affect the reliability of our attack is task scheduling. Specifically, if the thread under the SDT context (e.g., the worker thread) is de-scheduled by the OS while the attacking thread (e.g., the main thread) is executing, then the attack window will be increased. The only way to change the code cache's memory permission is through system calls, and a context switch is likely to happen during the system call. For example, the system call for changing memory access permissions on Linux is `mprotect`. During the invocation of `mprotect`, since we are using fine-grained protection, the virtual memory area needs to be split or merged. This will trigger the thread to be de-scheduled. As a result, the main thread (with higher priority than the worker) can gain control to launch attacks.

Considering these two factors, we tested our attack against the Chrome browser 100 times. Of these 100 tests, 91 succeeded.

3.5 System Design

This section presents the design of SDCG, which aims to achieve two goals: (1) SDCG should prevent all possible code cache injection attacks under our adversary model; and (2) SDCG should introduce acceptable performance overhead. Currently, SDCG is designed to be integrated with the targeted SDT, so we assume the source code of the SDT is available.

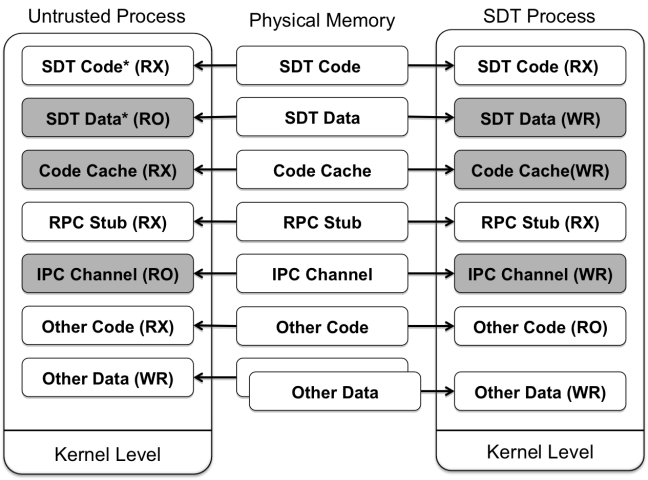

Figure 3: Overview of SDCG's multi-process-based architecture.

3.5.1 Overview and Challenges

Since the root cause of the attack is a writable code cache (either permanently or temporarily), we can prevent such attacks making one of two design choices: (1) ensure that no component other than the SDT can write to the code cache, e.g., through SFI or memory safety; or (2) ensure that the memory occupied by the code cache is always mapped as `RX`. We selected the second option for two reasons. First, we expect that the performance overhead of applying SFI or memory safety to a large, complex program (e.g., a web browser) would be very high. Second, implementing the first choice requires significant engineering effort.

Figure 3 shows the high level design of SDCG. The key idea is that through shared memory, the same memory content will be mapped into two (or more) different processes with different access permissions. In the untrusted process(es), the code cache will be mapped as `RX`; but in the SDT process, it will be mapped as `WR`. By doing so, SDCG prevents any untrusted code from modifying the code cache. At the same time, it allows the SDT to modify the code cache as usual. Whenever the SDT needs to be invoked (e.g., to install a new code fragment), the request will be served through a remote procedure call (RPC) instead of a normal function call.

To build and maintain this memory model, following technical and engineering challenges must be solved.

1. *Memory Map Synchronization.* Since the memory regions occupied by the code cache

are dynamically allocated and can grow and shrink freely, we need an effective way to dynamically synchronize memory mapping between the untrusted process(es) and the SDT process. More importantly, to make SDCG's protection mechanism work transparently, we have to make sure that the memory is mapped at exactly the same virtual address in all processes.

2. *Remote Procedure Call.* After relocating the SDT to another process, we need to make it remotely invocable by wrapping former local invocations with RPC stubs. Since RPC is expensive, we need to reduce the frequency of invocations, which also reduces the attack surface.

3. *Permission Enforcement.* Since SDCG's protection is based on memory access permissions, we must make sure that untrusted code cannot tamper with our permission scheme. Specifically, memory content can be mapped as either writable or executable, but never both at the same time.

3.5.2 Memory Map Synchronization

Synchronizing memory mapping between the untrusted process(es) and the SDT process is a bi-directional issue. On one side, when the SDT allocates a new code fragment in the SDT process, SDCG should map the same memory region in the untrusted process(es) at exactly the same address; otherwise the translated code will not work correctly (e.g., create an incorrect branching target). On the other side, the untrusted process may also allocate some resources that are critical to the SDT. For example, in the scenario of binary translation, when the untrusted process loads a dynamically linked module, SDCG should also load the same module at the same address in the SDT process; otherwise the SDT will not be able to locate the correct code to be translated. Moreover, this synchronization needs to be as transparent to the SDT as possible, so as to minimize code changes.

When creating shared memory, there are two possible strategies: on-demand and reservation-based. On-demand mapping creates the shared memory at the very moment a new memory region is required, e.g., when the SDT wants to add a new memory region to the code cache. However, as the process address space is shared by all modules of a program,

the expected address may not always be available in both the untrusted process and the SDT process. For this reason, we choose the reservation-based strategy. That is, when the process is initialized, we reserve (map) a large chunk of shared memory in both the untrusted process(es) and the SDT process. Later, any request for shared memory will be allocated from this shared memory pool. Note that in modern operating systems, physical memory resources are not mapped until the reserved memory is accessed, so our reservation-based strategy does not impose significant memory overhead.

Once the shared memory pool is created, synchronization can be done via inter-process communication (IPC). Specifically, when the SDT allocates a new memory region for the code cache, it informs the untrusted process(es) about the base address and the size of this new memory region. Having received this event, the untrusted process(es) maps a memory region with the same size at the same base address with the expected permission (`RX`). Similarly, whenever the untrusted process allocates memory that needs to be shared, a synchronization event is sent to the SDT process.

3.5.3 Remote Procedure Call

Writing RPC stubs for the SDT faces two problems: argument passing and performance. Argument passing can be problematic because of pointers. If a pointer points to a memory location that is different between the untrusted process and the SDT process, then the SDT ends up using incorrect data and causes run-time errors. Vice versa, if the returned value from the SDT process contains pointers that point to data not copied back, the untrusted code ends up running incorrectly. The common solution for the stub to serialize the object before passing it to the remote process instead of simply passing the pointer. Unfortunately, not all arguments have built-in serialization functionality. In addition, when an argument is a large object, performing serialization and copy for every RPC invocation introduces high performance overhead. Thus, in general, stub generation is not easy without support from the SDT or from program analysis.

To avoid this problem, SDCG takes a more systematic approach. Specifically, based on the observation that a majority of data that the SDT depends on is either read-only

or resides in dynamically mapped memory, we extend the shared memory to also include the dynamic data the SDT depends on. According to the required security guarantee, the data should be mapped with different permissions. By default, SDCG maps the SDT's dynamic data as read-only in the untrusted process, to prevent tampering by the untrusted code. However, if data-oriented attacks [41, 88, 89] are not considered, the SDT's dynamic data can be mapped as `WR` in the untrusted process. After sharing the data, we only need to handle a few cases where writable data (e.g., pointers within global variables) is not shared/synchronized.

Since RPC invocations are much more expensive than normal function calls, we want to minimize the frequency of RPC invocation. To do so, we take a passive approach. That is, we do not convert an entry from the SDT to RPC unless it modifies the code cache. Again, we try to achieve this goal without involving heavy program analysis. Instead, we leverage the regression tests that are usually distributed along with the source code. More specifically, we begin with no entries being converted to RPC and gradually convert them until all regression tests pass.

While our approach can be improved with more automation and program analysis, we leave these as future work because our main goal here is to design and validate that our solution is effective against the new code cache injection attacks.

3.5.4 Permission Enforcement

To enforce mandatory W⊕X, we leverage the delegation-based sandbox architecture [80]. Specifically, we intercept all system calls related to virtual memory management, and enforce the following policies in the SDT process:

(I) Memory cannot be mapped as both writable and executable.

(II) When mapping a memory region as executable, the base address and the size must come from the SDT process, and the memory is always mapped as `RX`.

(III) The permission of non-writable memory cannot be changed.

3.5.5 Security Analysis

In this section, we analyze the security of SDCG under our threat model. First, we show that our design can enforce permanent W⊕X policy. The first system call policy ensures that attackers cannot map memory that is both writable and executable. The second policy ensures that attackers cannot switch memory from non-executable to executable. The combination of these two policies guarantees that no memory content can be mapped as both writable and executable, either at the same time or alternately. Next, the last policy ensures that if there is critical data that the SDT depends on, it cannot be modified by attackers. Finally, since the SDT is trusted and its data is protected, the second policy can further ensure that only SDT-verified content (e.g., code generated by the SDT) can be executable. As a result, SDCG can prevent any code cache injection attack.

3.6 Implementation

We implemented two prototypes of SDCG, one for the Google V8 JS engine [85], and the other for the Strata DBT [169]. Both prototypes were implemented on Linux. We chose these two SDTs for the following reasons. First, JS engines are one of the most widely deployed SDTs. At the same time, they are also one of the most popular stepping stones for launching attacks. Among all JS engines, we chose V8 because it is open source, highly ranked, and there is a disclosed exploit [160]. Second, DBTs have been widely used by security researchers to build various security solutions [20, 142, 43, 90, 87]. Among all DBTs, we chose Strata because (1) it has been used to implement many promising security mechanisms, such as instruction set randomization [90], instruction layout randomization [87], etc; and (2) its academic background allowed us to have access to its source code, which is required for implementation of SDCG.

3.6.1 Shared Infrastructure

The memory synchronization and system call filtering mechanisms are specific to the target platform, but they can be shared among all SDTs.

Seccomp-Sandbox. Our delegation-based sandbox is built upon the seccomp-sandbox [83]

from Google Chrome. Although Google Chrome has switched to a less complicated process sandbox based on seccomp-bpf [52], we found that the architecture of seccomp-sandbox serves our goal better. Specifically, since seccomp only allows four system calls once enabled, and not all system calls can be fulfilled by the broker (e.g., `mmap`), the seccomp-sandbox introduced a trusted thread to perform system calls that cannot be delegated to the broker. To prevent attacks on the trusted thread, the trusted thread operates entirely on CPU registers and does not trust any memory that is writable to the untrusted code. When the trusted thread makes a system call, the system call parameters are first verified by the broker, and then passed through a shared memory that is mapped as read-only in the untrusted process. As a result, even if the other threads in the same process are compromised, they cannot affect the execution of the trusted thread. This provides a perfect foundation to securely build our memory synchronization mechanism and system call filtering mechanism.

To enforce the mandatory W⊕X policy, we modified the sandbox so that before entering sandbox mode, SDCG enumerates all memory regions and converts any `WRX` region to `RX`.

For RPC invocation, we also reused seccomp-sandbox's domain socket based communication channel. However, we did not leverage the seccomp mode in our current implementation for several reasons. First, it is not compatible with the new seccomp-bpf-based sandbox used in Google Chrome. Second, it intercepts too many system calls that are not required by SDCG. More importantly, both Strata and seccomp-bpf provide enough capability for system call filtering.

Shared Memory Pool. During initialization, SDCG reserves a large amount of consecutive memory as a pool. This pool is mapped as shared (`MAP_SHARED`), not file backed (`MAP_ANONYMOUS`) and with no permission (`PROT_NONE`). After this, any `mmap` request from the SDT allocates memory from this pool (by changing the access permission) instead of using the `mmap` system call. This guarantees any SDT allocated region can be mapped at exactly the same address in both the SDT process and the untrusted process(es).

After the sandbox is enabled, whenever the SDT calls `mmap`, SDCG generates a synchronized request to the untrusted process(es), and waits until the synchronization is done before returning to the SDT. In the untrusted process, the synchronization event is handled by the

trusted thread. It reads a synchronization request from the IPC channel and then changes the access permission of the given region to the given value. Since the parameters (base address, size and permission) are passed through the read-only IPC channel and the trusted thread does not use a stack, it satisfies our security policy for mapping executable memory.

Memory mapping in the untrusted process(es) is forwarded to the SDT process by the system call interception mechanism of the sandbox. The request first goes through system call filtering to ensure the security policy is enforced. SDCG then checks where the request originated. If the request is from the SDT, or is a special resource the SDT depends on (e.g., mapping new modules needs to be synchronized for Strata), the request is fulfilled from the shared memory pool. If it is a legitimate request from the untrusted code, the request is fulfilled normally.

System Call Filtering. SDCG rejects the following types of system calls.

- `mmap` with writable (`PROT_WRITE`) and executable (`PROT_EXEC`) permission.
- `mprotect` or `mremap` when the target region falls into a protected memory region.
- `mprotect` with executable (`PROT_EXEC`) permission.

SDCG maintains a list of protected memory regions. After the SDT process is forked, it enumerates the memory mapping list through `/proc/self/maps`, and any region that is executable is included in the list. During runtime, when a new executable region is created, it is added to the list; when a region is unmapped, it is removed from the list. If necessary, the SDT's dynamic data can also be added to this list.

For Strata, this filtering is implemented by intercepting the related system calls (`mmap`, `mremap`, and `mprotect`). For V8 (integrated with the Google Chrome browser), we rely on the seccomp-bpf filtering policies.

3.6.2 SDT Specific Handling

Next, we describe some implementation details that are specific to the target SDT.

Implementation for Strata. Besides the code cache, many Strata-based security mechanisms also involve some critical metadata (e.g., the key to decrypt a randomized

instruction set) that needs to be protected. Otherwise, attackers can compromise such data to disable or mislead critical functionalities of the security mechanisms. Thus, we extended the protection to Strata's code, data, and the binary to be translated. Fortunately, since Strata directly allocates memory from `mmap` and manages its own heap, this additional protection can be easily supported by SDCG. Specifically, SDCG ensures that all the memory regions allocated by Strata are mapped as either read-only or inaccessible. Note that we do not need to protect Strata's static data, because once the SDT process is forked, the static data is copy-on-write protected, i.e., while the untrusted code could modify Strata's static data, the modification cannot affect the copy in the SDT process.

Writing RPC stubs for Strata also reflects the differences in the attack model: since all dynamic data are mapped as read-only, any functionality that modified the data also needs to be handled in the SDT process.

Another special case for Strata is the handling of process creation, i.e., the `clone` system call. The seccomp-sandbox only handles the case for thread creation, which is sufficient for Google Chrome (and V8). But for Strata, we also need to handle process creation. The challenge for process creation is that once a memory region is mapped as shared, the newly created child process will also inherit this memory regions as shared. Thus, once the untrusted code forks a new process, this process also shares the same memory pool with its parent and the SDT process. If we want to enforce a $1:1$ serving model, we need to un-share the memory. Unfortunately, un-sharing memory under Linux is not easy: one needs to (1) map a temporary memory region, (2) copy the shared content to this temporary region, (3) unmap the original shared memory, (4) map a new shared memory region at exactly the same address, (5) copy the content back, and (6) unmap the temporary memory region. At the same time, the child process is likely to either share the same binary as its parent, which means it can be served by the same SDT; or call `execve` immediately after the fork, which completely destroys the virtual address space it inherited from its parent. For these reasons, we implemented a $N:1$ serving model for Strata, i.e., one SDT process serves multiple untrusted processes. The `clone` system call can then be handled in the same way for both thread creation and process creation. The only difference is that when a new memory region

is allocated from the shared memory pool, all processes need to be synchronized.

3.6.2.1 Implementation for V8

Compared with Strata, the biggest challenge for porting V8 to SDCG is the dynamic data used by V8. Specifically, V8 has two types of dynamic data: JS related data and its own internal data. The first type of data is allocated from custom heaps that are managed by V8 itself. Similar to Strata's heap, these heaps directly allocate memory from `mmap`, thus SDCG can easily handle this type of data. The difficulty comes from the second type of data, which is allocated from the standard C library (glibc on Linux). This makes it challenging to track which memory region is used by the JS engine. Clearly, we cannot make the standard C library allocate all the memory from the shared memory pool. However, as mentioned earlier in the design section, we have to share data via RPC and avoid serializing objects, especially C++ objects, which can be complicated. To solve this problem, we implemented a simple arena-based heap that is backed by the shared memory pool and modified V8 to allocate certain objects from this heap. Only objects that are involved in RPC need to be allocated from this heap, the rest can still be allocated from the standard C library.

Another problem is the stack. Strata does not share the same stack as the translated program, so it never reads data from the program's stack. This is not true for V8. In fact, many objects used by V8 are allocated on the stack. Thus, during RPC handling, the STD process may dereference pointers pointing to the stack. Moreover, since the stack is assigned during thread creation, it is difficult to ensure that the program always allocates stack space from our shared memory pool. As a result, we choose to copy stack content between the two processes. Fortunately, only 3 RPCs require a stack copy. Note that because the content is copied to/from the same address, when creating the trusted SDT process, we must assign it a new stack instead of relying on copy-on-write.

Writing RPC stubs for V8 is more flexible than Strata because dynamic data is not protected. For this reason, we would prefer to convert functions that are invoked less frequently. To achieve this goal, we followed two general principles. First, between the entry of the JS engine and the point where the code cache is modified, many functions could

be invoked. If we convert a function too high in the calling chain, and the function does not result in modification of the code cache under another context, we end up introducing unnecessary RPC overhead. For instance, the first time a regular expression is evaluated, it is compiled; but thereafter, the compiled code can be retrieved from the cache. Thus, we want to convert functions that are post-dominated by operations that modify the code cache. On the same time, if we convert a function that is too low in the calling chain, even though the invocation of this function always results in modification of the code cache, the function may be called from a loop, e.g., marking processes during garbage collection. This also introduces unnecessary overhead. Thus, we also want to convert functions that dominate as many modifications as possible. In our prototype implementation, since we did not use program analysis, these principles were applied empirically. In the end, we added a total of 20 RPC stubs.

3.7 Evaluation

To evaluate our prototype, we designed and performed experiments in order to answer the following questions:

- How effective is our design in blocking code cache injection attacks (§3.7.2)?
- How much overhead is incurred by our protection mechanism (§3.7.3 and §3.7.4)?

3.7.1 Setup

For our protected version of the Strata DBT, we measured the performance overhead using SPEC CINT 2006 [186]. Our protected version of the V8 JS engine was based on revision 16619. The benchmark we used to measure the performance overhead is the V8 Benchmark distributed with the source code (version 7) [162]. All experiments were run on a workstation with one Intel Core i7-3930K CPU (6-core, 12-thread) and 32GB memory. The operating system is the 64-bit Ubuntu 13.04 with kernel 3.8.0-35-generic.

3.7.2 Effectiveness

In §3.5.5, we provided a security analysis of our system design, which showed that if implemented correctly, SDCG can prevent all code cache injection attacks. In this section,

Table 1: RPC Overhead During the Execution of V8 Benchmark.

Benchmark	Avg Call Lat	Avg Ret Lat	# of Calls	w/ Stack Copy	w/o Stack Copy
Richards	4.70 μs	4.54 μs	1525	362 (23.74%)	1163 (76.26%)
DeltaBlue	4.28 μs	4.46 μs	2812	496 (17.64%)	2316 (82.36%)
Crypto	3.99 μs	4.28 μs	4596	609 (13.25%)	3987 (86.75%)
RayTrace	3.98 μs	4.00 μs	3534	715 (20.23%)	2819 (79.77%)
EarlyBoyer	3.87 μs	4.28 μs	5268	489 (9.28%)	4779 (90.72%)
RegExp	3.82 μs	5.06 μs	6000	193 (3.22%)	5807 (96.78%)
Splay	4.63 μs	5.04 μs	5337	1187 (22.24%)	5150 (77.76%)
NavierStokes	4.67 μs	4.82 μs	1635	251 (15.35%)	1384 (84.65%)

we evaluate our SDCG-ported V8 prototype to determine whether it can truly prevent the attack we demonstrated in §3.4.2.

The experiment was done using the same proof-of-concept code as described in §3.4.2. As the attack relies on a race condition, we executed it 100 times. For the version that is protected by naive W⊕X enforcement, the attack was able to inject shellcode into the code cache 91 times. For SDCG-protected version, all 100 attempts failed.

3.7.3 Micro Benchmark

The overhead introduced by SDCG comes from two major sources: RPC invocation and cache coherency.

RPC Overhead. To measure the overhead for each RPC invocation, we inserted a new field in the request header to indicate when this request was sent. Upon receiving the request, the handler calculates the time elapsed between this and the current time. Similarly, we also calculated the time elapsed between the sending and receiving of return values. To eliminate the impact from cache synchronization, we pinned all threads (in both the untrusted process and the SDT process) to a single CPU core. The frequency of RPC invocation also effects overall overhead, so we also collected this number during the evaluation.

Table 1 shows the result from the V8 benchmark, using the 64-bit release build. The average latency for call request is around 3-4 μs and the average latency for RPC return is around 4-5 μs. Thus, the average latency for an RPC invocation through SDCG's communication channel is around 8-9 μs. The number of RPC invocations is between 1,525 and 6,000. Since the input is fixed, this number is stable, with small fluctuations caused by garbage collection. Compared to the overall overhead presented in the next subsection, it

follows the expected pattern that the larger the number of RPC invocations, the higher the overhead is. Among all RPC invocations, less than 24% require a stack copy.

Cache Coherency Overhead. SDCG involves at least three concurrently running threads: the main thread in the untrusted process, the trusted thread in the untrusted process, and the main thread in the SDT process. This number can increase if the SDT to be protected already uses multiple threads. On a platform with multiple cores, these threads can be scheduled to different cores. Since SDCG depends heavily on shared memory, OS scheduling for these threads can also affect performance, i.e., cache synchronization between threads executing on different cores introduces additional overhead.

In this subsection, we report this overhead at the RPC invocation level. In the next subsection, we present its impact on overall performance. The evaluation also uses V8 benchmark. To reduce the possible combination of scheduling, we disabled all other threads in V8, leaving only the aforementioned three threads. The Intel Core i7-3930K CPU on our testbed has six cores. Each core has a dedicated 32KB L1 data cache and 256KB integrated L2 cache. A 12MB L3 cache is shared among all cores. When Hyperthreading is enabled, each core can execute two concurrent threads.

Given the above configuration, we have tested the following scheduling:

1. All threads on a single CPU thread (affinity mask = {0});
2. All threads on a single core (affinity mask = {0,1});
3. Two main threads that frequently access shared memory on a single CPU thread, trusted thread freely scheduled (affinity mask = {0},{*});
4. Two main threads on a single core, trusted thread freely scheduled (affinity mask = {0,1},{*});
5. All three threads on different cores (affinity mask = {0},{2},{4}); and
6. All three threads freely scheduled (affinity mask = {*},{*},{*}).

Table 2 shows the result, using the 64-bit release build. All the numbers are for RPC invocation, with return latency omitted. Based on the result, it is clear that scheduling

Table 2: Cache Coherency Overhead Under Different Scheduling Strategies.

Benchmark	Schedule 1	Schedule 2	Schedule 3	Schedule 4	Schedule 5	Schedule 6
Richards	4.70 μs	13.76 μs	4.47 μs	14.25 μs	12.85 μs	13.37 μs
DeltaBlue	4.28 μs	13.29 μs	4.31 μs	13.85 μs	14.09 μs	15.84 μs
Crypto	3.99 μs	10.91 μs	3.98 μs	14.07 μs	12.47 μs	13.48 μs
RayTrace	3.98 μs	14.99 μs	4.05 μs	14.76 μs	13.15 μs	12.35 μs
EarlyBoyer	3.87 μs	13.70 μs	3.87 μs	14.27 μs	13.42 μs	13.47 μs
RegExp	3.82 μs	14.64 μs	3.85 μs	14.48 μs	13.55 μs	12.32 μs
Splay	4.63 μs	12.92 μs	4.49 μs	13.22 μs	13.36 μs	15.11 μs
NavierStokes	4.67 μs	12.06 μs	4.47 μs	13.02 μs	14.80 μs	12.65 μs

Table 3: SPEC CINT 2006 Results.

Benchmark	Native	Strata	SDCG (Pinned)	SDCG (Free)
perlbench	364	559	574	558
bzip2	580	600	613	602
gcc	310	403	420	410
mcf	438	450	479	471
gobmk	483	610	623	611
hmmer	797	777	790	777
sjeng	576	768	784	767
libquantum	460	463	511	474
h264ref	691	945	980	971
omnetpp	343	410	450	428
astar	514	546	587	563
xalancbmk	262	499	515	504
GEOMEAN	461	566	592	576

has a great impact on the RPC latency. If the two main threads are not scheduled on the same CPU thread, the average latency can exacerbate to 3x-4x slower. On the other hand, scheduling for the trusted thread has little impact on the RPC latency. This is expected because the trusted thread is only utilized for memory synchronization.

3.7.4 Macro Benchmark

Since OS scheduling can have a large impact on performance, for each benchmark suite, we evaluated two CPU schedules. The first (*Pinned*) pins both the main threads from the untrusted process and the SDT process to a single core; and the second (*Free*) allows the OS to freely schedule all threads.

SPEC CINT 2006. Both the vanilla Strata and the SDCG-protected Strata are built as 32-bit. The SPEC CINT 2006 benchmark suite is also compiled as 32-bit. Since all benchmarks from the suite are single-threaded, the results of different scheduling strategies only reflect the overhead caused by SDCG.

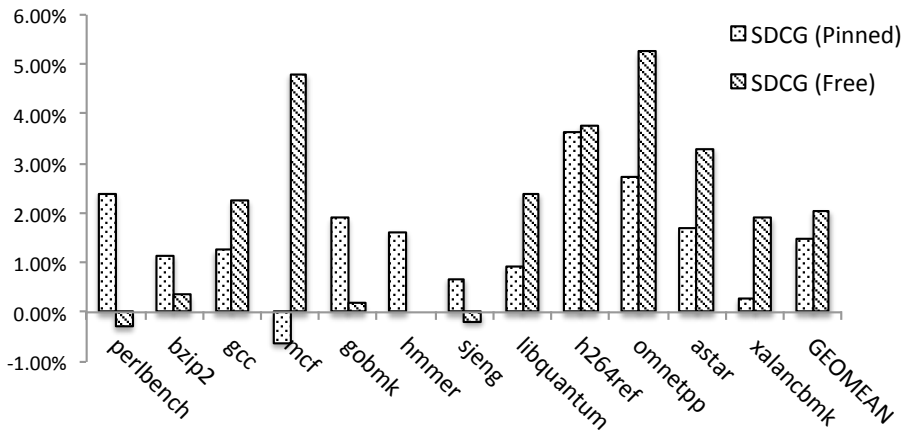

Figure 4: SPEC CINT 2006 Slowdown of Strata.

Table 3 shows the evaluation result. The first column is the result of running *natively*. The second column is the result for Strata without SDCG. We use this as the baseline for calculating the slowdown introduced by SDCG. The third column is the result for SDCG with pinned schedule, and the last column is the result for SDCG with free schedule. Since the standard deviation is small (less than 1%), we omitted this information.

The corresponding slowdown is shown in Figure 4. For all benchmarks, the slowdown introduced by SDCG is less than 6%. The overall (geometric mean) slowdown is 1.46% for the pinned schedule, and 2.05% for the free schedule.

Since SPEC CINT is a computation-oriented benchmark suite and Strata does a good job reducing the number of translator invocations, we did not observe a significant difference between the pinned schedule and the free schedule.

JavaScript Benchmarks. Our SDCG-protected V8 JS engine was based on revision 16619. For better comparison with an SFI-based solution [9], we performed the evaluation on both IA32 and x64 release builds. The arena-based heap we implemented was only enabled for SDCG-protected V8. To reduce the possible combination of scheduling, we also disabled all other threads in V8.

Table 4 shows the results for the IA32 build, and Table 5 shows the results for the x64 build. The first column is the baseline result; the second column is the result of SDCG-protected V8 with a pinned schedule; and the last column is the result of SDCG-protected V8

Table 4: V8 Benchmark Results (IA32).

Benchmark	Baseline	SDCG (Pinned)	SDCG (Free)
Richards	24913 (2.76%)	23990 (0.28%)	24803 (1.72%)
DeltaBlue	25657 (3.31%)	24373 (0.43%)	25543 (3.86%)
Crypto	20546 (1.61%)	19509 (1.27%)	19021 (1.95%)
RayTrace	45399 (0.38%)	42162 (0.75%)	43995 (6.46%)
EarlyBoyer	37711 (0.61%)	34805 (0.27%)	34284 (0.82%)
RegExp	4802 (0.34%)	4251 (1.04%)	2451 (3.82%)
Splay	15391 (4.47%)	13643 (0.71%)	9259 (8.18%)
NavierStokes	23377 (4.15%)	22586 (0.42%)	23518 (1.26%)
Score	21071 (0.72%)	19616 (0.35%)	17715 (1.86%)

Table 5: V8 Benchmark Slowdown (x64).

Benchmark	Baseline	SDCG (Pinned)	SDCG (Free)
Richards	25178 (3.39%)	24587 (2.31%)	25500 (3.24%)
DeltaBlue	24324 (3.65%)	23542 (0.38%)	24385 (2.54%)
Crypto	21313 (3.16%)	20551 (0.26%)	20483 (2.57%)
RayTrace	35298 (5.97%)	32972 (1.03%)	35878 (1.66%)
EarlyBoyer	32264 (4.42%)	30382 (0.61%)	30135 (1.04%)
RegExp	4853 (3.59%)	4366 (0.82%)	2456 (7.72%)
Splay	13957 (6.02%)	12601 (2.92%)	7332 (9.85%)
NavierStokes	22646 (2.48%)	21844 (0.30%)	21468 (3.45%)
Score	19712 (3.57%)	18599 (0.62%)	16435 (1.03%)

with a free schedule. All results are the geometric mean over 10 executions of the benchmark. The number in the parentheses is the standard deviation as a percentage. As we can see, the fluctuation is small, with the baseline and a free schedule slightly higher than a pinned schedule.

The corresponding slowdown is shown in Figure 5 (for IA32 build) and Figure 6 (for x64 build). Overall, we did not observe a significant difference between the IA32 build and the x64 build. For four benchmarks (Richards, DeltaBlue, Crypto, and NavierStokes), the slowdown introduced by SDCG is less than 5%, which is negligible because they are similar to the standard deviation. The other four benchmarks (RayTrace, EarlyBoyer, RegExp, and Splay) have higher overhead, but with a pinned schedule, the slowdown is within 11%, which is much smaller than previous SFI-based solutions [9] (79% on IA32).

There are two major sources of overhead. For RPC overhead, we can see a clear trend that more RPC invocations (Table 1) implies more slowdown. However, the impact of cache coherency overhead caused by different scheduling strategies is not consistent. For some benchmarks (Richards, DeltaBlu, and RayTrace), free scheduling is faster than pinned

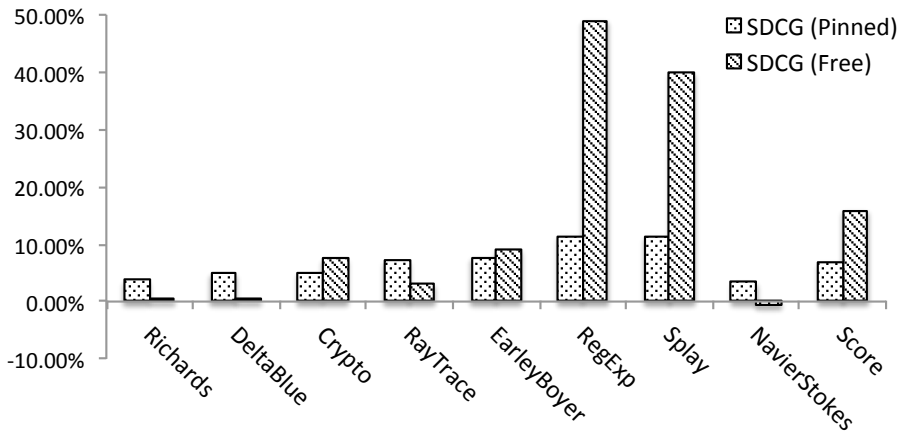

Figure 5: V8 Benchmark Slowdown (IA32).

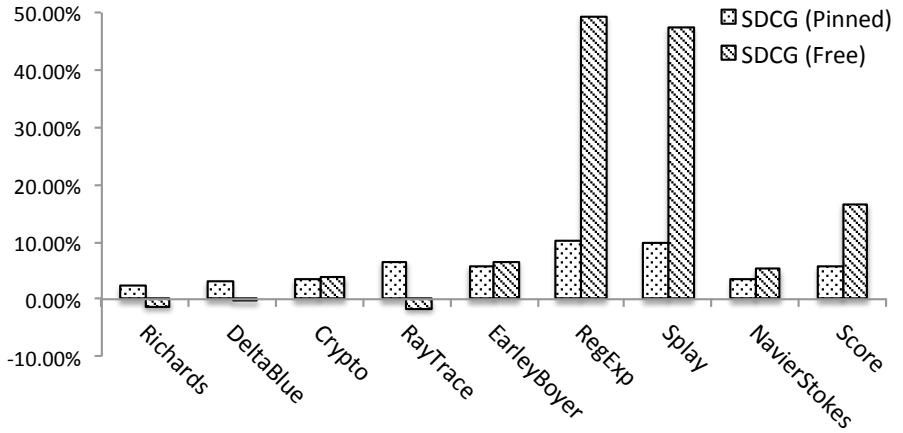

Figure 6: V8 Benchmark Slowdown (x64).

scheduling; for some benchmarks (Crypto and EarlyBoyer), overhead is almost the same; but for two benchmarks (RegExp and Splay), the overhead under free scheduling is much higher than pinned scheduling. We believe this is because these two benchmarks depend more heavily on shared data (memory) access. Note that, unlike Strata, for SDCG-protected V8, we not only shared the code cache, but also shared the heaps used to store JS objects, for the ease of RPC implementation. Besides RPC frequency, this is another reason why we observed a higher overhead compared with SDCG-protected Strata.

3.8 Limitations and Future Work

This section discusses the limitations of this work and potential future work.

3.8.1 Reliability of Race Condition

Although we only showed the feasibility of the attack in one scenario, the dynamic translator can be invoked under different situations, each of which has its own race condition window. Some operations can be quick (e.g., patching), while others may take longer. By carefully controlling how the translator is invoked, we can extend the race condition window and make such attacks more reliable.

In addition, OS scheduling can also affect the size of the attack window. For example, as we have discussed in §3.4, the invocation of `mprotect` is likely to cause the thread to be swapped out of the CPU, which will extend the attack window.

3.8.2 RPC Stub Generation

To port a dynamic translator to SDCG, our current solution is to manually rewrite the source code. Although the modification is relatively small compared to the translator's code size, the process still requires the developer to have a good understanding of the internals of the translator. This process can be improved or even automated through program analysis. First, our current RPC stub creation process is not sound, i.e., we heavily relied on the test input. Thus, if a function is not invoked during testing, or the given parameter does not trigger the function to modify the code cache, then we miss this function. Second, to reduce performance overhead and the attack surface, we want to create stubs only for functions that (1) are post-dominated by operations that modify the code cache; and (2) dominate as many modification operations as possible. Currently, this is done empirically. Through program analysis, we could systematically and more precisely identify these "key" functions. Finally, for the ease of development, our prototype implementation uses shared memory to avoid deep copy of objects when performing RPC. While this strategy is convenient, it may introduce additional cache coherency overhead. With the help of program analysis, we could replace this strategy with object serialization, but only for data that is accessed during RPC.

3.8.3 Performance Tuning

In our current prototype implementations, the SDTs were not aware of our modification to their architectures. Since their optimization strategy may not be ideal for SDCG, it is possible to further reduce the overhead by making the SDT be aware of our modification. First, one major source of SDCG's runtime overhead is RPC invocation, and the overhead can be reduced if we reduce the frequency of code cache modification. This can be accomplished in several ways. For instance, we can increase the threshold to trigger code optimization, use more aggressive speculative translation, separate the garbage collection, etc.

Second, in our implementations, we used the domain socket-based IPC channel from the seccomp-sandbox. This means for each RPC invocation, we need to enter the kernel twice; and both the request/return data need to be copied to/from the kernel. While this approach is more secure (in the sense that a sent request cannot be maliciously modified), if the request is always untrusted, then using a faster communication channel (e.g., ring buffer) could further reduce the overhead.

Third, we used the same service model as seccomp-sandbox in our prototypes. That is, RPC requests are served by a single thread in the SDT process. This strategy is sufficient for SDTs where different threads share the same code cache (e.g., Strata) since modifications need to be serialized anyway to prevent a data race condition. However, this service model can become a bottleneck when the SDT uses different code caches for different thread (e.g., JS engines). For such SDTs, we need to create dedicated service threads in the SDT process to serve different threads in the untrusted process.

In addition, our current prototype implementations of SDCG are not hardware-aware. Different processors can have different shared cache architectures and cache management capabilities, which in turn affects cache synchronization between different threads. Specifically, on a multi-processor system, two cores may or may not share the same cache. As we have demonstrated, if the translator thread and the execution thread are scheduled to two cores with different cache, then the performance is much worse than when they are scheduled to cores with the same cache. To further reduce the overhead, we can assign processor affinity according to the hardware features.

3.9 Summary

In this chapter, we highlighted that a code cache injection attack is a viable exploit technique that can bypass many state-of-the-art defense mechanisms. To defeat this threat, we proposed SDCG, a new architecture that enforces mandatory W⊕X policy. To demonstrate the feasibility and benefit of SDCG, we ported two software dynamic translators, Google V8 and Strata, to this new architecture. Our development experience showed that SDCG is easy to adopt and our performance evaluation showed the performance overhead is small.

CHAPTER IV

PREVENTING KERNEL PRIVILEGE ESCALATION ATTACKS WITH DATA-FLOW INTEGRITY

4.1 Motivation

The operation system (OS) kernel is often the de facto trusted computing base (TCB) of the whole software stack, including many higher level security solutions. For example, the security of an application/browser sandbox usually depends on the integrity of the kernel. Unfortunately, kernel vulnerabilities are not rare in commodity OS kernels like Linux, Windows, and XNU. Once the kernel is compromised, attackers can bypass any access control checks, escalate their privileges, and hide the evidence of attacks. For example, to prevent remote attacks, modern browsers all isolate the potential vulnerable rendering engine in a sandbox. Theoretically, this should prevent all attacks as even if attackers can compromise the rendering engine, they cannot perform any malicious activity. However, due to kernel vulnerabilities, attackers usually can find a way to break the sandbox and take out the attacks, which has been demonstrated repeatedly in the Pwn2Own competitions.

Among all kernel vulnerabilities, the ones related to memory corruption are the most prevalent because all commodity kernels are implemented in low-level unsafe language like C and assembly. Memory corruption bugs are also the most dangerous ones because they can grant attackers great capabilities. For these reasons, most kernel attacks exploit memory corruption vulnerabilities.

Existing solutions to this problem can be classified into two main categories: off-line tools and runtime protection mechanisms. Off-line tools [206, 111, 37, 220, 99] try to identify potential kernel memory safety violations so that developers can fix them before deployment. Although these tools have successfully found many vulnerabilities in the kernel, they have limitations. First, most bug-finding tools tend to generate lots of false positives, which makes it hard for developers to filter out the real bugs. Second, tools that can prove an

implementation is free of memory safety issues usually do not scale well. This means they can only be applied to small, self-contained kernel extensions [138, 19] or micro-kernels [107].

For runtime protection mechanisms, as discussed in §2 a majority of them focus on protecting the control flow. Many mechanisms are proposed to protect code integrity and stop malicious kernel extensions from loading [173]. Others focus on preventing control-flow hijacking attacks, such as `ret2usr` [155, 104] and return-oriented programming (ROP) [115]. More recently, researchers have demonstrated the feasibility of enforcing control-flow integrity (CFI) in kernel space [209, 56]. However, in addition to problems discovered in CFI [32, 77, 82, 66], the more fundamental problem is that, even with perfect CFI, many vulnerabilities are still exploitable. This is because OS kernels are mainly data-driven, so CFI can be easily bypassed by data-oriented attacks [41]. For example, to bypass discretionary access control (DAC), attackers just need to overwrite the subject's identity of the current process with the one of the root/administrators.

Some technologies are capable of preventing data-oriented attacks. For example, software fault isolation (SFI) [207, 74, 35, 123] can be used to isolate small "untrusted" modules from tampering the core kernel components. However, a recent study on Linux kernel vulnerabilities [38] has shown that vulnerabilities in the core components are as common as vulnerabilities in third-party drivers. The secure virtual architecture [57] is able to provide full memory safety for the kernel, but its performance overhead is too high to be deployed in practice.

Another promising technique that can prevent both control-flow and data-oriented attacks is data-flow integrity (DFI) [34]. Similar to CFI, DFI guarantees that runtime data-flow cannot deviate from the data-flow graph generated from static analysis. For example, data from a string buffer should never flow to the return address on stack (control-data), or to the `uid` (non-control-data). Utilizing this technique, we can enforce a large spectrum of security invariants in the kernel to defeat different attacks. For instance, to defeat privilege escalation attacks, we can utilize DFI to enforce security invariants that are related to kernel access control mechanisms (a.k.a. reference monitors). Specifically, assuming a reference monitor is implemented correctly, its runtime correctness relies on three high-level security

invariants [7]:

I **Complete mediation**: attackers should not be able to bypass any access control check; and

II **Tamper proof**: attackers should not be able to tamper with the integrity of either the code or data of the reference monitor.

While this approach sounds intuitive at high level, enforcing it with practical runtime performance overhead is challenging. First, unlike kernel extensions, which are usually self-contained and use dedicated data structures, access control checks are scattered throughout the kernel, and related data are mixed with other data. Therefore, the protection technique must be deployed kernel-wide. Moreover, without hardware support, software-based DFI implementation can be very expensive, e.g., the original DFI enforcement implementation imposed an average 104% overhead for user-mode CPU benchmarks [34]. Since OS kernels tend to be more data intensive than those CPU benchmarks, we expect the same technique will be even more expensive for kernel protection.

To overcome these challenges, we propose KENALI, a system that is both principled and practical. KENALI consists of two key techniques. Our first technique, INFERDISTS, is based on the observation that although the protection has to be kernel-wide, only a small portion of data is essential for enforcing the two security invariants related to access control. For ease of discussion, we refer to this set of data as *distinguishing regions* (formally defined in §4.6.1). Hence, instead of enforcing DFI for all kernel data, we only need to enforce DFI over the distinguishing regions. Our second technique, PROTECTDISTS, is a new technique to enforce selective DFI over the distinguishing regions. In particular, since distinguishing regions only constitutes a small portion of all kernel data, PROTECTDISTS uses a two-layer protection scheme. The first layer provides a coarse-grained but low-overhead data-flow isolation that prevents illegal data-flow from non-distinguishing regions to distinguishing regions. After this separation, the second layer then enforces fine-grained DFI over the distinguishing regions. Combining these two techniques, namely, INFERDISTS and PROTECTDISTS, KENALI is able to enforce the two security invariants without sacrificing too much performance.

4.2 Threat Model and Assumptions

Our research addresses kernel attacks that exploit memory corruption vulnerabilities to achieve privilege escalation. We only consider attacks that originate from unprivileged code, such as user-space application without root privilege. Accordingly, attacks that exploit vulnerabilities in privileged system processes (e.g., system services or daemons) are out-of-scope. Similarly, because kernel drivers and firmware are already privileged, we do not consider attacks that originate from kernel rootkits and malicious firmware. Please note that we do *not* exclude attacks that exploit vulnerabilities in kernel drivers, but only attacks from malicious kernel rootkits. Other kernel exploits such as denial-of-service (DoS), logical/semantic bugs, or hardware bugs [106] are also out-of-scope. We believe this is a realistic threat model because (1) it covers a majority of the attack surface and (2) techniques to prevent the excluded attacks have been proposed by other researchers and can be combined with KENALI.

However, we assume a *powerful* adversary model for memory corruption attacks. Specifically, we assume there is one or more kernel vulnerabilities that allow attackers to read and write word-size value at an arbitrary virtual address, as long as that address is mapped in the current process address space.

Since many critical non-control-data are loaded from disk, another way to compromise the kernel is to corrupt the disk. Fortunately, solutions already exist for detecting and preventing such attacks. For example, secure boot [10] can guarantee that the kernel and KENALI are not corrupted on boot; SUNDR enables data to be stored securely on untrusted servers [116]; ZFS enforces end-to-end data integrity using the Merkle tree [26]; and Google also introduced a similar mechanism called DMVerity [8] to protect the integrity of critical disk partitions. For these reasons, we limit the attack scope of this paper to memory-corruption-based attacks and exclude disk-corruption-based attacks.

4.3 Related work

In this section, we compare KENALI with a variety of defense mechanisms to defend memory-corruption-based attacks.

4.3.1 Kernel Integrity

Early work on runtime kernel integrity protection focused on code integrity, including boot time [10] and post boot [173, 16, 62]. KENALI also needs to guarantee kernel code integrity, and our approach is similar to [62]. After rootkit became a major threat to the kernel, techniques have also been proposed to detect malicious modifications to the kernel [158, 159, 18, 31]. Compared to these works, KENALI has two differences: (1) threat model: rootkit are code already with kernel privilege, and their goal is to hide their existence; and (2) soundness: most of these tools are not sound on identifying all critical data (i.e., have false negatives), but our approach is sound.

4.3.2 Software Fault Isolation

Because many memory corruption vulnerabilities are found in third-party kernel drivers due to relatively low code quality, many previous works focused on confining the memory access capabilities of these untrusted code with SFI [207, 74, 35, 123]. A major limitation of SFI, as pointed out by [38], is that the core kernel may also contain many bugs, which cannot be handled by SFI.

4.3.3 Data-flow Integrity

DFI is a more general technique than SFI, as it can mitigate memory corruptions from any module of the target program/kernel. KENALI differs from previous work on DFI enforcement [4, 34] in the following aspects. First, KENALI is designed for OS kernels, while previous work focused on user-mode programs; so KENALI requires additional care for the kernel environment. Second, previous work protects all program data, while KENALI aims at reducing the overhead by only enforcing DFI over a small portion of data that are critical to security invariants. Finally, as the effectiveness of DFI also depends on the quality of the point-to analysis, KENALI leveraged a more precise context-sensitive point-to analysis tailored for the kernel [31].

4.3.4 Dynamic Taint Analysis

DTA is similar to DFI and has also been used to prevent attacks [142]. The main difference is, in DTA, data provenance is defined by high-level abstractions like file, network, and syscall; but in DFI, data provenance is defined by the instruction that writes to the memory. For this reason, DTA is usually much more expensive than DFI, as it also needs to track the data provenance/tag for computation operations.

4.3.5 Memory Safety

Besides DFI, another important defense technique is runtime memory safety enforcement. Theoretically, *complete* memory safety enforcement is more precise than DFI because it is based on concrete runtime information, but its performance overhead is also higher. For example, SVA [57] imposes a 2.31x - 13x overhead on LMbench. Generally, because the result generated by InferDists is orthogonal to the runtime enforcement techniques, it can also be combined with memory safety. In this work, we chose DFI because we found that most accesses to distinguishing regions are safe; so DFI requires fewer checks and no metadata propagation for pointer assignments and arithmetic. However, if the overhead for memory safety enforcement becomes reasonable, e.g., with hardware assistant like Intel MPX [93], we can also switch the second layer protection from DFI to memory safety.

A recent memory safety work [108] demonstrated that, to defeat a certain type of attack (control-flow hijacking), it is sufficient to only protect a portion of data, thus reducing the performance overhead. While Kenali leveraged a similar idea, it addressed an important yet previously unsolved problem—what data are essential to prevent privilege escalation attacks.

4.3.6 Control-flow Integrity

Because a majority of real-world attacks are control-flow hijacking, CFI [1] has been a hot topic in recent year and has been demonstrated to be extensible to the kernel [56]. However, as discussed in §4.4, non-control-data attacks are both feasible and capable enough for a full privilege escalation attack. Furthermore, as CFI becomes more practical, attackers are

also likely to move to non-control-data attacks. For these reasons, we believe KENALI makes valuable contributions, as (1) it can prevent both control-data and non-control-data attacks; and (2) compared to previous discussions on non-control-data attacks [41, 88], it provides an automated and systematic way to discover critical non-control-data.

4.4 Demonstration Attacks

We use a real vulnerability, CVE-2013-6282 [131], as a running example to demonstrate the various types of attacks feasible under our threat model, and to illustrate how and why these attacks can bypass the state-of-the-art defense techniques like CFI and ad-hoc kernel integrity protection. Given this vulnerability, we begin with an existing attack against the Linux kernel. Then, step-by-step, we further demonstrate two additional attacks showing how this original attack can be extended to accomplish a full rooting attack.

4.4.1 Simple rooting attacks

CVE-2013-6282 allows attackers to read and write arbitrary kernel memory, which matches our adversary model. The corresponding rooting attack provides a good example of how most existing kernel privilege escalation exploits work:

a) Retrieving the address of `prepare_kernel_cred()` and `commit_creds()`. Depending on the target system, they can be at fixed addresses, or obtainable from the kernel symbol table (`kallsyms_addresses`);

b) Invoking `prepare_kernel_cred()` and pass the results to `commit_creds()`, then the kernel will replace the credential of the current thread with one of root privilege.

Step b can be done in several ways: attackers can overwrite a function pointer to a user mode function that links these two functions together (i.e., `ret2usr`). Alternatively, attackers can also link them through return-oriented programming.

4.4.2 Bypassing CFI with non-control-data attacks

The above attack can be prevented by kernel-wide CFI [56]. But CFI can be easily bypassed by non-control-data attacks: by locating the `cred` structure and overwriting the `euid` field,

attackers can still escalate the privilege to the root user. The `cred` structure can be located in many ways: (1) if `kallsyms` is available and contains the address of `init_task`, we can easily traverse the process list to locate the `task_struct` of the current process, then `task_struct->cred`; (2) if there is a vulnerability that leaks the stack address (e.g., CVE-2013-2141), attackers can directly obtain the address of the `thread_info` structure, then follows the links to locate the `task_struct`; and (3) with arbitrary memory read capability, attackers can also scan the whole kernel memory and use signature matching to identify the required data structures [117]. (4) alternatively, they can also start with some fixed global objects and traverse the memory using technique presented in [31].

4.4.3 Bypassing CFI with control-data attacks

In this example, we designed a new attack to demonstrate another limitation of CFI. Specifically, to prevent root privilege from being acquired through compromising system daemons, Android leverages SELinux, a mandatory access control mechanism, to further restrict the root privilege [180]. Therefore, disabling SELinux is a necessary step to gain the full root privilege. This can be achieved through control-data attacks that do not violate CFI. In particular, SELinux callbacks are stored in a dispatch table that has a special initialization phase:

```
1  // @security/capability.c
2  void security_fixup_ops(struct security_operations *ops) {
3    if (!ops->sb_mount)
4      ops->sb_mount = cap_sb_mount;
5    if (!ops->capable)
6      ops->capable = cap_capable;
7    ...
8  }
9  static int cap_sb_mount(const char *dev_name, ...) {
10     return 0;
11 }
```

Basically, if a Linux Security Module (LSM) does not implement a hook, its callback function will be set to the default one (e.g., `cap_sb_mount`). Therefore, the default callback functions are also valid control transfer targets, but they usually perform no checks before

directly returning 0. Based on this observation, SELinux can then be disabled without violating CFI, by setting every callback function pointer to its default one.

4.4.4 Diversity of non-control-data attacks

Some existing kernel integrity protection mechanisms also try to protect non-control-data [158, 159, 18], such as `uid`. However, these approaches are inherently limited because there can be many different non-control-data involved in access control. For example, from the target kernel we evaluated, we found that 2,419 data structures contain critical data (§4.8). Here, we use a concrete non-control-data attack to demonstrate this limitation of previous work.

Specifically, the above two steps only grant attackers temporary root privilege until the next reboot (a.k.a. tethered root). To acquire permanent root privilege (untethered root), the de facto way is to install the `su` utility. To do so, however, there is one more protection to bypass: read-only mounting. That is, to protect critical system files, the system partition on most Android devices is mounted as read-only. Existing rooting attacks achieve this goal by remounting the partition as writable, but we achieve this goal through another non-control-data attacks:

```
1 // @fs/namespace.c
2 int __mnt_is_readonly(struct vfsmount *mnt) {
3     if (mnt->mnt_flags & MNT_READONLY)
4         return 1;
5     if (mnt->mnt_sb->s_flags & MS_RDONLY)
6         return 1;
7     return 0;
8 }
```

As we can see, by overwriting data fields like `mnt_flags` and `s_flags`, attackers can bypass the read-only mount and overwrite the system partition.

4.5 Technical Approach

Figure 7 provides an overview of our technical approach, which consists of two steps. In the first step, we use a novel program analysis technique InferDists to systematically infer a complete and minimized set of distinguishing regions. In the second step, we employ a

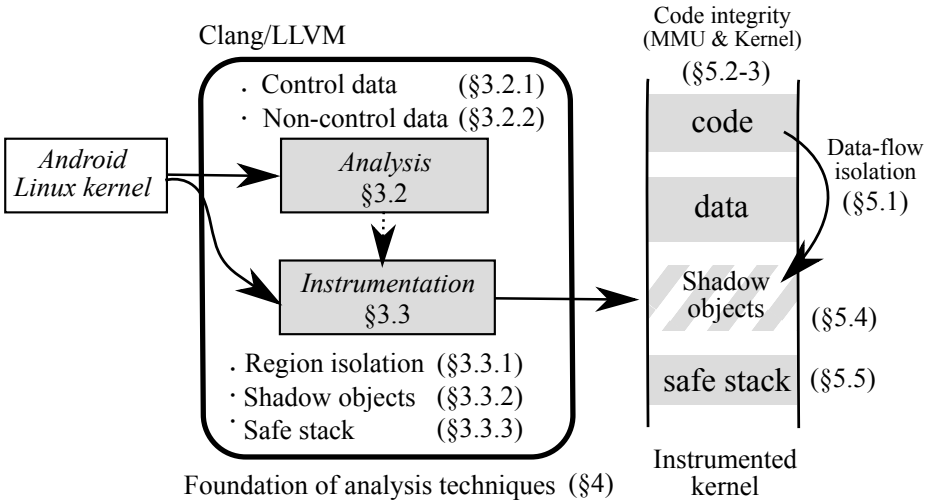

Figure 7: Overview of KENALI's approach.

lightweight runtime protection technique, PROTECTDISTS, to enforce DFI over the inference result.

4.5.1 Inferring Distinguishing Regions

In this subsection, we present our approach to the inference problem. The challenge is that for security, our solution must be sound (i.e., no false negatives), but for performance, we want the size of the inference result to be as small as possible.

Control-Data. As discussed in §4.1, INVARIANT I can be violated via control-data attacks. Therefore, all control-data must be included as distinguishing regions so as to enforce CFI. Control-data in the kernel has two parts: general control-data and kernel-specific data. General control-data (e.g., function pointers) can be identified based on the type information [108]. Kernel-specific data, such as interrupt dispatch table, have been enumerated in [56]. Since our approach to infer these data does not differ much from previous work, we omit the details here.

Non-Control-Data. The challenge for inferring distinguishing non-control-data is the soundness, i.e., whether a proposed methodology can discover *all* distinguishing regions. To address this challenge, we developed INFERDISTS, an automated program analysis technique. The key idea is that access controls are implemented as security checks, and while a kernel may have many security checks scattered throughout different components, they all follow

one consistent semantic: *if a security check fails, it should return a security related error codes*. For example, a POSIX-compatible kernel returns `-EACCES` (permission denied) [194] to indicate the current user does not have access to the requested resources. Similarly, Windows also has `ERROR_ACCESS_DENIED` [191]. Leveraging this observation, InferDists is able to collect security checks without manual annotation. Then, distinguishing regions can be constructed via standard dependency analysis over the conditional variables of security checks. Next, we use Example 1 as a running example to demonstrate how InferDists works. The formal model of InferDists and proof for its soundness are provided in §4.6.

```
1  int acl_permission_check(struct inode *inode, int mask) {
2    unsigned int mode = inode->i_mode;
3    if (current_cred->fsuid == inode->i_uid)
4      mode >>= 6;
5    else if (in_group_p(inode->i_gid))
6      mode >>= 3;
7    if ((mask & ~mode &
8        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
9      return 0;
10   return -EACCES;
11 }
```

Example 1: A simplified version of the discretionary access control (DAC) check in the Linux kernel.

In step **(1)**, InferDists collects the return instructions `R` that may return an error code that we are interested in.

> In Example 1, the function returns `-EACCES` at line 10, so InferDists include this instruction into `R`.

In step **(2)**, InferDists collects the branch instructions `I` that determine if a run either executes a return instructions in `R` or performs a potentially privileged operation. The conditional variables `B` of `I` will be distinguishing variables of this function.

> In Example 1, the condition of the `if` statement at line 7 is the distinguishing conditional variable of the function. Note, the `if` statement at line 3 is not included because it is post-dominated by the `if` statement at line 7.

It is worth mentioning that we need to handle some special cases in step **(2)**. First, in some cases, when a security check fails, it will not directly return an error but will continue to another alternative check, e.g., in the `setuid` system call, three checks are performed:

```
1 if (!nsown_capable(CAP_SETUID) &&
2     !uid_eq(kuid, old->uid) && !uid_eq(kuid, new->suid))
3     return -EPERM;
```

In this case, when the first check `nsown_capable` failed, the control flow will go to the second, and possibly the third check. Although these are all security checks, in the control flow graph (CFG), only the branch instruction for the last `uid_eq` check will lead to an error return. If we only consider this branch, then we will miss the first two checks. Therefore, we must also consider branch(es) that dominate a security check. However, naively including all dominators will introduce many false positives. To reduce false positives, InferDists conservatively excludes two cases: (a) a branch can lead to non-related error return (e.g., `-EINVAL`) and (b) a branch instruction is post-dominated by either a security check or checks in (a), i.e., diamond-shaped nodes.

In step **(3)**, InferDists collects and returns all memory regions that have dependencies on conditional variables in `B`. For completeness, we consider both *data- and control-dependencies*, and the analysis is inter-procedural and iterative, i.e., we perform the analysis multiple times until there is no new data get included.

> In Example 1 the `if` statement on line 7 has two conditional variables, `mask` and `mode`. Data-dependency over `mode` would include `i_mode` (line 2) and control-dependency would include `i_uid`, `fsuid` (line 3), and the return value of `in_group_p` (line 5). Because our dependency analysis is inter-procedural, we will also include `i_gid`.

Sensitive Pointers. Pointers to sensitive regions must be protected as well; otherwise, attackers can *indirectly* control the data in distinguishing regions by manipulating these pointers [88]. For instance, instead of overwriting the `euid` field of a `cred` structure, an attacker could overwrite the `task_struct->cred` pointer to point to a `cred` structure whose `euid == 0`.

Hence, after collecting control- and non-control-data, the analysis collects the *sensitive pointers* of a program, and includes such pointers in distinguishing regions. A pointer is sensitive if it points to a data structure that contains distinguishing regions. There are two

points worth noting: (1) sensitive pointers are defined recursively and (2) even with the existence of generic pointers, we still can collect a sound over-approximation set of sensitive pointers using a static program analysis, so no false negatives will be introduced.

4.5.2 Protecting Distinguishing Regions

After collecting the distinguishing regions, the next step is to enforce DFI over the inference result. The challenge for this step is how to minimize the performance overhead on commodity processors that lack support for fine-grained data-flow tracking. To address this challenge, our key observation is that, after (conceptually) separating the memory into distinguishing and non-distinguishing regions, there could be three types of data-flow: (1) within non-distinguishing regions, (2) between two regions, and (3) within distinguishing regions. Our goal is to prevent attackers from introducing *illegal data-flow* to compromise distinguishing regions. Obviously, it is not possible to compromise distinguishing regions based on the first type of data-flow, but most legal data-flows actually belong to this type. Therefore, if we purely rely on DFI to vet all data-flows, there will be a huge number of unnecessary checks. Based on this observation, we design a two-layer scheme: the first layer is a lightweight data-flow isolation mechanism that prevents illegal data-flow of the second type; then we use the more expensive DFI enforcement to prevent illegal data-flow of the third type. With this two-layer approach, we can reduce the performance overhead without sacrificing security guarantees.

Data-Flow Isolation. There are two general approaches to enforce data-flow isolation: software-based and hardware-based. Software fault isolation is not ideal because it requires instrumenting a majority of write operations. For example, in our prototype implementation, only 5% of write operations can access distinguishing regions; thus, relying on SFI would end up with instrumenting the remaining 95% of write operations. At the same time, not all hardware isolation mechanisms are equally efficient. Since distinguishing regions are actually not continuous, but interleaved with non-distinguishing regions, it would be ideal if the hardware could support fine-grained memory isolation. Unfortunately, most commodity hardware does not support this but only provides coarse-grained isolation mechanisms. Based

on their corresponding overhead (from low to high), the available options on commodity hardware are: the segmentation on x86-32 [223], the execution domain on ARM-32 [235], the WP flag on x86-64 [209, 62], hardware virtualization [178, 157], and TrustZone [16].

In this work, we explored the feasibility of hardware-based data-flow isolation for AArch64, which is becoming more and more popular for mobile devices, but has not been well studied before. For AArch64, most of the aforementioned features are not available, except TrustZone[1]; but world switch for TrustZone is usually very expensive because it needs to flush the cache and sometimes the TLB (translation look-aside buffer) too. To solve this problem, we developed a novel, virtual address space-based isolation mechanism. Specifically, to reduce the overhead of context switching between different virtual address spaces, modern processors usually tag the TLB with an identifier associated with the virtual address space. Utilizing this feature, we can create a trusted virtual address space by reserving an identifier (e.g., ID = 0). By doing so, context switching between the untrusted context and trusted context becomes less expensive because it requires neither TLB flush nor cache flush (see §4.7 for more details).

Write Integrity Test. In addition to preventing illegal data-flow from non-distinguishing regions to distinguishing regions, we use DFI to prevent illegal data-flow within distinguishing regions. However, instead of checking data provenance at read, we leveraged the write integrity test (WIT) technique [4]. We chose this technique for the following reasons. First, we found that the memory access pattern for distinguishing regions is very asymmetric, e.g., reading `uid` is prevalent, but updating `uid` is very rare. Thus, by checking write operations, we can reduce the number of checks that need to be performed. Second, WIT reasons about safe and unsafe write operations and only instruments unsafe writes, which matches another observation—the majority of writes to the distinguishing regions are safe and do not require additional checks. Finally, compared to memory safety enforcement techniques [57], WIT is less expensive because it does not require tracking pointer propagation. Because of page limitation, we omit the details of WIT; please refer to the original paper for more details.

[1]Hardware virtualization extension is defined, but a majority of processors do not implement it; and for those who have this feature, the bootloader usually disabled it.

However, in order to apply this technology to the kernel, we made one change. In particular, the original WIT implementation used a context-sensitive field-insensitive point-to analysis, but since OS kernels usually contain a lot of generic pointers and linked lists, we replaced the point-to analysis with a context-sensitive and field-sensitive analysis that is tailored for the kernel [31].

Shadow Objects. Shadow objects is a work-around for the lack of fine-grained hardware isolation mechanism. Specifically, as a hardware protection unit (e.g., page) may contain both distinguishing and non-distinguishing regions, once we write-protect that page, we also have to pay additional overhead for accessing non-distinguishing regions. One solution to this problem is to manually partition data structures that contain mixed regions into two new data structures [185]. However, this approach does not scale and requires heavy maintenance if the data structure changes between different kernel versions. Our solution to this problem is *shadow objects*, i.e., if a kernel object contains both regions, then we will create two copies of it—a normal copy for the non-distinguishing regions and a shadow copy for the distinguishing regions. Shadow memory may consume up to two times the original memory, but because commodity OS kernels usually use memory pools (e.g., `kmem_cache`) to allocate kernel objects, it allows us to reduce the memory overhead by dedicating different pools to objects that need to be shadowed. This nice feature also allows us to eliminate maintaining our own metadata for shadow objects allocation/free; and to perform fast lookup—giving a pointer to normal object, its shadow object can be acquired by adding a fixed offset.

Safe Stack. Similar to heap, stack also needs a "shadow" copy for the lack of fine-grained isolation. However, stack is treated differently for its uniqueness. First, stack contains many critical data that are not visible at the source code level, such as return addresses, function arguments, and register spills [51]. Second, the access pattern of these critical data is also different: write accesses are almost as frequent as read accesses. For these reasons, we leveraged the safe-stack technique proposed in [108], with a few improvements to make this technique work better for the kernel. First, we used more precise inter-procedural analysis to reduce the number of unsafe stack objects from 42% to 7%. Second, as the number of

unsafe stack objects is very small, instead of using two stacks, we keep only one safe stack and move all unsafe objects to the heap to avoid maintaining two stacks.

4.6 Formal Model

To demonstrate that our technical approach is correct, we formalize the problem of preventing privilege escalation via memory corruption and describe an approach to solve the problem. In §4.6.1, we formulate the problem of rewriting a monitor with potential memory vulnerabilities to protect privileged system resources. In §4.6.2, we formulate the sub-problem of inferring a set of memory regions that, if protected, are sufficient to ensure that a monitor protects privileged resources. In §4.6.3, we formulate the problem of rewriting a program to protect a given set of memory regions. In §4.6.4, we show that our approaches to solve the inference and protection problems can be composed to solve the overall problem.

4.6.1 Problem Definition

A language of monitors. A monitor state is a valuation of data variables and address variables, where each address is represented as a pair of a base region and an offset. Let the space of *machine words* be denoted Words and let the space of *error codes* be denoted $\mathsf{ErrCodes} \subseteq \mathsf{Words}$. Let the space of *memory regions* be denoted $\mathsf{Regions}$, and let an *address* be a region paired with a machine word, i.e., $\mathsf{Addrs} = \mathsf{Regions} \times \mathsf{Words}$.

Let the space of *control locations* be denoted $\mathtt{Locs}$, let the space of *protection colors* be denoted colors, let the space of data variables be denoted $\mathtt{D}$, and let the space of address variables be denoted $\mathtt{A}$.

Definition 1 *A monitor* state *consists of (1) a control location, (2) a map from each address to its size, (3) a map from each address to its protection color, and (4)–(6) maps from* $\mathtt{D}$*,* $\mathtt{A}$*, and* Addrs *to the word values that they store. The space of monitor states is denoted* Q*.*

A monitor *instruction* is a control location paired with an operation. The set of operations contains standard read operations $\mathtt{read\ a, d}$, conditional branches $\mathtt{bnz\ d}$, returns $\mathtt{ret\ d}$,

arithmetic operations, and logical operations. The set of operations also contains the following non-standard operations:

(1) For each protection color $c \in \mathsf{colors}$, each address variable $\mathtt{a} \in \mathtt{A}$, and each data variable $\mathtt{d} \in \mathtt{D}$, $\mathtt{alloc}[c]\ \mathtt{a}, \mathtt{d}$ allocates a new memory region $R \in \mathsf{Regions}$, sets the size of R to be to the word stored in $\mathtt{d}$, and stores the address $(R, 0)$ in $\mathtt{a}$. The space of allocation instructions is denoted Allocs.

(2) For each protection color $c \in \mathsf{colors}$, each data variable $\mathtt{d} \in \mathtt{D}$, and each address variable $\mathtt{a} \in \mathtt{A}$, $\mathtt{write}[c]\ \mathtt{d}, \mathtt{a}$ attempts to write the value stored in $\mathtt{d}$ to the address stored in $\mathtt{a}$. Let $\mathtt{a}$ store the address (R, o). If in the current state, the color of R is c, then the write occurs; otherwise, the program aborts. If $\mathtt{a}$ stores an address inside of its base region, then the write is *safe*; otherwise, the write is *unsafe*. The space of write instructions is denoted Writes.

Definition 2 *For each instruction* $\mathtt{i} \in \mathsf{Instrs}$*, the* transition relation *of* $\mathtt{i}$ *is denoted* $\sigma[\mathtt{i}] \subseteq Q \times Q$*.*

We refer to σ as the *unrestricted* transition relation, as it places no restriction on the target address of a write outside of a region (in contrast to *restricted* transition relations, defined in Defn. 5). The formal definitions of the transition relations of each instruction follow directly form the instruction's informal description and thus are omitted.

Definition 3 *A* monitor *is a pair* $(\mathtt{I}, \mathtt{A})$*, where* $\mathtt{I} \in \mathsf{Instrs}^*$ *is a sequence of instructions in which (1) each variable is defined exactly once and (2) only the final instruction is a return instruction;* $\mathtt{A} \subseteq \mathsf{Allocs}$ *are the allocation sites of* privileged regions*. The space of monitors is denoted* $\mathsf{Monitors} = \mathsf{Instrs}^* \times \mathcal{P}(\mathsf{Instrs})$*.*

A run *of M is an alternating sequence of states and instructions such that adjacent states are in the transition relation of the unrestricted semantics of the neighboring instruction; the runs of* $\mathtt{M}$ *under the unrestricted semantics are denoted* $\mathsf{Runs}(\mathtt{M})$*. The* safe *runs of* $\mathtt{M}$ *are the runs in which each write instruction only writes within its target region.*

A monitor $\mathtt{M}' \in \mathsf{Monitors}$ *is a* refinement *of* $\mathtt{M}$ *if each run of* $\mathtt{M}'$ *is a run of* $\mathtt{M}$*.* $\mathtt{M}'$ *is a* non-blocking refinement *of* $\mathtt{M}$ *if (1)* $\mathtt{M}'$ *is a refinement of* $\mathtt{M}$ *and (2) each safe run of* $\mathtt{M}$ *is a*

run of $\mathtt{M}'$.

Monitor consistency. A monitor *inconsistency* is a pair of runs (r_0, r_1) from the same initial state, where r_0 accesses a privileged region and r_1 returns an error code.

A monitor is *weakly consistent* if for each *run*, the monitor exclusively either accesses privilege regions or returns an error code. That is, monitor $\mathtt{M} = (\mathtt{I}, \mathtt{A}) \in \mathsf{Monitors}$ is weakly consistent if there is no run $r \in \mathsf{Runs}(\mathtt{M})$ such that (r, r) is an inconsistency. A core assumption of our work, grounded in our study of kernel access control mechanisms, is that practical monitors are usually written to be weakly consistent.

A monitor is *strongly consistent* if for each *initial state*, the monitor exclusively either accesses privileged regions or returns an error code. That is, $\mathtt{M}$ is *strongly consistent* if there are no runs $r_0, r_1 \in \mathsf{Runs}(\mathtt{M})$ such that (r_0, r_1) is an inconsistency.

The consistent-refinement problem. Each strongly consistent monitor is weakly consistent. However, a monitor $\mathtt{M}$ may be weakly consistent but *not* strongly consistent if it contains a memory error that prohibits $\mathtt{M}$ from ensuring that all runs from a given state either access privileged regions or return error codes. The main problem that we address in this work is to instrument all such monitors to be strongly consistent.

Definition 4 *For monitor* $\mathtt{M}$*, a solution to the* consistent-refinement problem Refine(M) *is a non-blocking refinement of* M *(Defn. 3) that is strongly consistent.*

We have developed a program rewriter, Kenali, that attempts to solve the consistent-refinement problem. Given a monitor $\mathtt{M}$, Kenali first infers a set of *distinguishing allocation sites* $\mathtt{A}$ for which it is sufficient to protect the integrity of all regions in order to ensure consistency (§4.6.2). Kenali then rewrites $\mathtt{M}$ to protect the integrity of all regions allocated at $\mathtt{A}$ (§4.6.3). The rewritten module $\mathtt{M}'$ is a non-blocking refinement of $\mathtt{M}$, and all potential inconsistencies of $\mathtt{M}'$ can be strongly characterized (§4.6.4).

4.6.2 Inferring distinguishing regions

Problem formulation. We now formulate the problem of inferring a set of memory regions that are sufficient to protect a monitor to be strongly consistent. We first define a

semantics for monitors that is parameterized on a set of allocation sites $\mathtt{A}$, under which the program can only modify a region allocated at a site in $\mathtt{A}$ with a safe write.

Definition 5 *For allocation sites* $\mathtt{A} \subseteq \mathsf{Allocs}$ *and instruction* $\mathtt{i} \in \mathsf{Instrs}$*, the* restricted semantics of R $\sigma_{\mathtt{A}}(\mathtt{i}) \subseteq Q \times Q$ *is the transition relation over states such that:* ***(1)*** *if* $\mathtt{i}$ *is not a write, then* $\sigma_{\mathtt{A}}[\mathtt{i}]$ *is identical to* $\sigma[\mathtt{i}]$*;* ***(2)*** *if* $\mathtt{i}$ *is a write, then for an unsafe write, the program may write to any region* not allocated at a site in $\mathtt{A}$

For each monitor $\mathtt{M} \in \mathsf{Monitors}$*, the runs of* $\mathtt{M}$ *under the restricted semantics for* R *are denoted* $\mathsf{Runs}_R(\mathtt{M})$*.*

The distinguishing-site inference problem is to infer a set of allocation sites $\mathtt{A}$ such that if all regions allocated at sites in $\mathtt{A}$ are protected, then the monitor is consistent.

Definition 6 *For each monitor* $\mathtt{M} = (\mathtt{I}, \mathtt{A}) \in \mathsf{Monitors}$ *and set of regions, a solution to the* distinguishing-site inference *problem* $\textsc{Dists}(\mathtt{M})$ *is a set of allocation sites* $\mathtt{A}' \subseteq \mathsf{Allocs}$ *such that* $\mathtt{M}$ *is consistent under the restricted semantics* $\mathsf{Runs}_{\mathtt{A}'}(\mathtt{M})$*. We refer to such a set* $\mathtt{A}'$ *as* distinguishing sites *for* $\mathtt{M}$*.*

Inferring distinguishing sites. In this section, we present a solver, InferDists, for solving Dists. InferDists proceeds in three steps: **(1)** InferDists collects the return instructions $\mathtt{R}$ that may return an error code. **(2)** InferDists collects the condition variables $\mathtt{B}$ of the branch instructions that determine if a run either executes a return instructions in $\mathtt{R}$ or accesses *any* memory region. **(3)** InferDists returns the dependency sites of all condition variables of $\mathtt{B}$. We now describe phases in detail.

Background: data-dependency analysis. For monitor $\mathtt{M}$, data variable $\mathtt{x} \in \mathtt{D}$, and allocation site $\mathtt{a} \in \mathsf{Allocs}$, $\mathtt{a}$ is a *dependency site* of $\mathtt{x}$ if over some run of $\mathtt{M}$, the value stored in some region allocated at $\mathtt{a}$ partially determines the value stored in $\mathtt{x}$ (the formal definition of a data dependency is standard) [3]. The data-dependency-analysis problem is to collect the dependency sites of a given set of data variables.

Definition 7 *For a monitor* $\mathtt{M} \in \mathsf{Monitors}$ *and data variables* $\mathtt{D} \subseteq \mathtt{D}$*, a solution to the* data-dependency-analysis *problem* $\textsc{Deps}(\mathtt{M}, \mathtt{D})$ *is a set of allocation sites that contain the dependency sites of all variables in* $\mathtt{D}$*.*

Our solver for the DISTS problem, named INFERDISTS, constructs instances of the DEPS problem and solves the instances by invoking a solver, INFERDEPS. The implementation of INFERDEPS used by INFERDISTS is a field-sensitive and context-sensitive analysis based on previous work [31].

Phase 1: collect error-return instructions. Phase 1 of INFERDISTS collects an over-approximation $\texttt{E}$ of the set of instructions that may return error codes. While in principle the problem of determining whether a given control location returns an error code is undecidable, practical monitors typically use simple instruction sequences to determine the codes that may be returned by a given instruction. Thus, INFERDISTS can typically collect a precise set $\texttt{R}$ using constant folding, a standard, efficient static analysis [3].

Phase 2: collect distinguishing condition variables. After collecting an over-approximation $\texttt{E}$ of the instructions that may return error codes, INFERDISTS collects an over-approximation $\texttt{C}$ of data variables that determine if a program returns an error code or accesses a sensitive resource; we refer to such data variables as *distinguishing condition variables*. INFERDISTS collects $\texttt{C}$ by performing the following steps: **(1)** INFERDISTS computes the post-dominator relationship $\mathsf{PostDom} \subseteq \mathsf{Instrs} \times \mathsf{Instrs}$, using a standard efficient algorithm. **(2)** INFERDISTS computes the set of pairs $\mathsf{ImmPred} \subseteq \mathsf{Instrs} \times \mathsf{Instrs}$ such that for all instructions $\texttt{i}, \texttt{j} \in \mathsf{Instrs}$, $(\texttt{i}, \texttt{j}) \in \mathsf{ImmPred}$ if there is a path from $\texttt{i}$ to $\texttt{j}$ that contains no post-dominator of $\texttt{i}$. $\mathsf{ImmPred}$ can be computed efficiently from $\texttt{M}$ by solving standard reachability problems on the control-flow graph of $\texttt{M}$. **(3)** INFERDISTS collects the set of branch instructions $\texttt{B}$ such that for each branch instruction $\texttt{b} \equiv \texttt{bnz x}, \texttt{T}, \texttt{F} \in \texttt{B}$, there is some error-return instruction $\texttt{e} \in \texttt{E}$ and some access $\texttt{a} \in \mathsf{accesses}$ such that $\mathsf{ImmPred}(\texttt{b}, \texttt{T})$ and $\mathsf{ImmPred}(\texttt{b}, \texttt{F})$. **(4)** INFERDISTS returns all condition variables of instructions in $\texttt{B}$.

Phase 3: collect data dependencies of conditions. After INFERDISTS collects a set of distinguishing condition variables $\texttt{C}$, it collects the dependency sites $\texttt{A}'$ of $\texttt{C}$ by invoking INFERDEPS (Defn. 7) on $\texttt{M}$ and $\texttt{C}$.

4.6.3 Protecting distinguishing regions

The region-protection problem. For monitor M and set of allocation sites A ⊆ Allocs, the region-protection problem is to color M so that it protects each region allocated at each site in A.

Definition 8 *For each monitor* M ∈ Monitors *and set of allocation sites* A ⊆ Allocs*, a solution to the* distinguishing-site-protection problem DistsProt(M, A) *is a monitor* M′ ∈ Monitors *such that each run of* M′ *under the unrestricted semantics (Defn. 2) is a run of* M *under the restricted semantics for* A *(Defn. 5).*

Background: writes-to analysis. For module M ∈ Monitors, the writes-to analysis problem is to determine, for each `write` instruction w in a monitor, the set of regions that w may write to in some run of M.

Definition 9 *For monitor* M ∈ Monitors*, a solution to the* writes-to analysis problem WrTo(M) *is a binary relation* $R \subseteq$ Writes × Allocs *such that if there is some run of* M *in which a write-instruction* w ∈ Writes *writes to a region allocated at allocation site* a ∈ Allocs*, then* (w, a) ∈ R*.*

Our implementation of ProtectDists uses a writes-to analysis SolveWrTo, which is built from a points-to analysis provided in the LLVM compiler framework [119].

A region-protecting colorer. Given an input monitor M ∈ Monitors and a distinguishing set of allocation sites A ⊆ Allocs, ProtectDists solves the protection problem DistsProt(M, A) using an approach that is closely related to the WIT write-integrity test [4]. In particular, ProtectDists performs the following steps: **(1)** ProtectDists obtains a writes-to relation W by invoking SolveWrTo on the writes-to analysis problem WrTo(M). **(2)** ProtectDists constructs the restriction of W to only allocation sites in A, denoted $W_{\mathtt{A}}$. **(3)** ProtectDists collects the set $\mathcal{C}$ of connected components of W, and for each component $C \in \mathcal{C}$, chooses a distinct color c_C. **(3)** ProtectDists replaces each allocation instruction `alloc`[0] a, d of M in component $C \in \mathcal{C}$ with the "colored" instruction `alloc`[c_C] a, d. **(4)**

ProtectDists replaces each write instruction $\texttt{write}[0]\ \texttt{a}, \texttt{d}$ of $\texttt{M}$ in component $D \in \mathcal{C}$ with the "colored" instruction $\texttt{write}[c_D]\ \texttt{a}, \texttt{d}$.

ProtectDists may thus be viewed as a "paramaterized WIT" that uses WIT's technique for coloring write-instructions and regions to protect only the subset of the regions that a monitor may allocate to determine whether to access a privileged region or return an error code.

4.6.4 Protected monitors as refinements

In this section, we characterize the properties of modules instrumented by solving the distinguishing-site inference (§4.6.2) and protection problems (§4.6.3). Let the module instrumenter Kenali be defined for each monitor $\texttt{M} \in \mathsf{Monitors}$ as follows:

$$\textsc{Kenali}(\texttt{M}) = \textsc{ProtectDists}(\texttt{M}, \textsc{InferDists}(\texttt{M}))$$

Kenali does not instrument a monitor to abort unnecessarily on runs in which the monitor does not perform an unsafe write.

Theorem 1 *For each monitor* $\texttt{M} \in \mathsf{Monitors}$, Kenali(M) *is a non-blocking refinement of* M.

While the modules generated by Kenali may have access-control inconsistencies, each inconsistency can be characterized by the results of the points-to analysis used by ProtectDists.

Theorem 2 *For each monitor* $\texttt{M} \in \mathsf{Monitors}$, *let* W *be the writes-to relation of* M *found by* SolveWrTo *(Defn. 9). If* (r_0, r_1) *is an inconsistency of* Kenali(M), *then* r_0 *is of the form* $q_0, \ldots, q_n, (\texttt{w} \equiv \texttt{write a}, \texttt{d}), q_{n+1}$, *and there are a pair of regions* $R_0 \neq R_1 \in \mathsf{Regions}$ *allocated at sites* $\texttt{A}_0, \texttt{A}_1 \in \mathsf{Allocs}$ *such that (1)* R_0 *is the target region of* w, *(2)* R_1 *is the region written by* w, *and (3)* $\texttt{A}_0$ *and* $\texttt{A}_1$ *are in the writes-to set for* w *in* W.

Our practical evaluation of our approach (§4.8.2) indicates that the accuracy of state-of-the-art points-to analyses greatly restricts the potential for inconsistencies in rewritten programs.

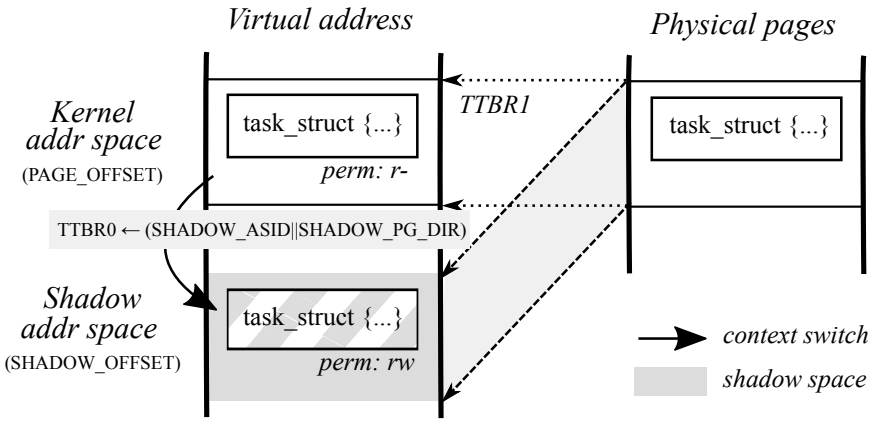

Figure 8: Shadow address space of KENALI.

4.7 A Prototype for Android

In this section, we present the prototype implementation for KENALI. As a demonstration, we focus on the AArch64 Linux kernel that powers Android devices. We chose AArch64 Linux for the following reasons. First, INFERDISTS requires source code to perform the analysis, and thanks to the success of Android, Linux is the most popular open-sourced kernel now. Second, due to the complexity of the Android ecosystem, a kernel vulnerability usually takes a long cycle to patch [203]. This means kernel defense techniques play an even more critical role for Android. Third, among all the architectures that Android supports, ARM is the most popular. Moreover, compared to other architectures like x86-32 [223] x86-64 [62] and AArch32 [235], there is little research on efficient isolation techniques for AArch64. Despite this choice, we must emphasize that our techniques are general and can be implemented on other hardware architectures and even other OS kernels (see §4.9).

We start this section with a brief introduction of the AArch64 virtual memory system architecture (VMSA) and how we leveraged various hardware features to implement data-flow isolation (§4.7.1). Next, we describe the techniques we used to enforce MMU integrity (§4.7.2). Then we discuss the challenges we had to overcome to implement the shadow objects technique on top of the Linux SLAB allocator (§4.7.3). Finally, we describe our implementation of safe stack for this particular prototype (§4.7.4).

4.7.1 Data-flow Isolation

For this prototype, we developed a new virtual address space isolation-based technique that leverages several nice features of the AArch64 virtual memory system architecture (VMSA) [12].

AArch64 VMSA. The AArch64 supports a maximum of 48-bit virtual address (VA), which is split into two parts: the bottom part is for user space, and the top part is for kernel space. VA to PA (physical address) translation descriptors are configured through two control registers: `TTBR0` (translation table base register) for the bottom half and `TTBR1` for the top half.

To minimize the cost of context switch, the AArch64 TLB has several optimizations. First, it allows global pages (mainly for kernel), i.e., translation results are always valid regardless of current address space. Second, each process-specific VA is associated with an ASID (address space identifier), i.e., translation results are cached as (ASID + VA) $\Rightarrow$ PA. Third, if the hardware virtualization extension is enabled, every VA is also tagged with a VMID (virtual machine identifier). In our target platform, the hardware supports 16-bit ASID and the ASID of current process is configured to be stored in `TTBR0` registers.

Shadow Address Space. Our data-flow isolation implementation leverages the ASID tagging feature and is based on shadow address space, as illustrated in Figure 8. Under this isolation scheme, the same physical page is mapped into two different address spaces with different access permissions. Although the shadow address space technique is not new, the key advantage of our approach is that it does not require TLB flush for context switch. More specifically, most previous systems map the same physical page at the *same VA*. However, because kernel memory is mapped as global, context switching always requires TLB flush to apply the new permission. Our approach instead maps the same physical memory at two *different VAs*. This allows us to leverage the ASID tagging feature to avoid TLB flush for context switch. Another benefit of this approach is that since it does not change the access permissions of global pages, it is *not* subject to multi-core-based attacks, i.e., when one core un-protects the data, attackers can leverage another core to attack.

In the shadow address space, we map the whole physical memory to a fixed virtual address (`SHADOW_OFFSET`) as read-writable. But in the original logical mapping (`PAGE_OFFSET`), memory may be mapped with more restricted permissions. For example, pages used for distinguishing regions are mapped as read-only. And the reason we choose to map the shadow address space at fixed VAs is to make the address space conversion between shadow, virtual and physical as efficient as possible.

Atomic Primitive Operations. Apparently we cannot keep the shadow address space always available; otherwise attackers can just write to the shadow address. Therefore, we only switch to the shadow address space whenever necessary and immediately switch back to the original (user) context after the operation is done. Moreover, since the kernel can be preemptive (as our target kernel), we also need to disable interruption under the shadow address space to prevent the operation from being interrupted. Otherwise, it could provide attackers chances to launch attacks or break the context, i.e., after exception handling, the shadow address space will not be restored, so writing to that VA will either crash the kernel or trash the user space data. For these reasons, we want to make every operation under the shadow address space atomic and as simple as possible. Currently, we support the following atomic primitive operations: write a single 8-, 16-, 32-, and 64-bit data, `memcpy`, and `memset`. Example 2 gives an example of performing an atomic 64-bit write operation in the shadow address space.

```
1  ; performing *%addr = %value
2  mrs     x1, daif                        ; save IRQ state
3  msr     daifset, #2                     ; disable IRQ
4  mrs     x2, ttbr0_el1                   ; save current ttbr0
5  msr     ttbr0_el1, %shadow_pgd_and_asid ; context switch
6  str     %value, %addr                   ; update shadow object
7  dmb     ishst                           ; store barrier
8  msr     ttbr0_el1, x2                   ; restore ttbr0
9  isb                                     ; instruction barrier
10 msr     daif, x1                        ; restore IRQ
```

Example 2: An example of performing an atomic 64-bit write operation in the shadow address space.

4.7.2 MMU Integrity

Since our isolation is based on virtual address space, we must guarantee that attackers cannot compromise our isolation scheme. We achieve this goal by enforcing three additional security invariants:

III. **MMU isolation**: We enforce that only MMU management code can modify MMU-related data, including hardware configuration registers and page tables.

IV. **Code integrity**: We enforce that attackers cannot modify existing kernel code or launch code injection attacks.

V. **Dedicated entry and exit**: We enforce that MMU management code is always invoked through dedicated entries and exits, so that attackers cannot jump to the middle of an MMU function and launch deputy attacks.

MMU Isolation. Our enforcement technique for INVARIANT III is similar to HyperSafe [209] and nested kernel [62]. First, we enforce that only MMU management code can modify MMU-related configuration registers. Compare to other platforms, enforcing this policy is relatively easier for AArch64: (1) because AArch64 uses one single instruction set that is fix-sized and well aligned, we statically verify that no other code can modify MMU configuration registers; (2) for configuration registers whose values do not need to be changed, such as `TTBR1`, we enforce that they are always loaded with constant values; (3) for registers that can be re-configured like `SCTLR`, we enforce that the possible values either provide the same minimal security guarantees (e.g., enabling and disabling `WXN` bit does not affect the protection because page tables can override) or crash the kernel (e.g., since the kernel VA is much higher than PA, disabling paging will crash the kernel).

The second step is to enforce that all memory pages used for page tables are mapped as read-only in logical mapping. This is done as follows. (1) We break down the logical mapping from section granularity (2MB) to page granularity (4K), so that we can enforce protection at page-level. (2) All physical pages used for initial page tables (i.e., logical mapping, identical mapping and shadow address space) are all allocated from a dedicated

area in the `.rodata` section. This is possible because the physical memory for most mobile devices cannot be extended. (3) After kernel initialization, we make all initial page tables as read-only in the logical mapping. (4) We enforce that physical pages used for these critical page tables can never be remapped, nor can their mapping (including access permissions) be modified. This is possible because kernel is always loaded at the beginning of the physical memory according to the ELF section order, so physical pages used for these critical data will have their frame number smaller than the end of the `.rodata` section. Specifically, we enforce that for any physical page whose page frame number is smaller than the end of `.rodata` section cannot be remapped nor can its mapping be modified. (5) Any memory page allocated by the MMU management code is immediately marked as read-only after receiving it from the page allocator.

Code Integrity. Enforcing kernel code integrity is essential for enforcing DFI; otherwise attackers can disable our protection by either removing our instrumentations or injecting their own code. We achieve this goal by enforcing two page table invariants. First, similar to page tables, we enforce that the kernel code (`.text`) section is always mapped as read-only. Second, we enforce that except for the `.text` section, no memory can be executable with kernel privilege, i.e., always has `PXN` (privilege execution never) bit [12] set.

Dedicated Entries and Exits. Enforcing dedicated entries and exits of MMU management code is trivial, as KENALI protects all code pointers, i.e., its capability of defeating control-flow hijacking attacks is equivalent to CPI [108], which is equivalent to fine-grained CFI [1].

4.7.3 Shadow Objects

Shadow object support includes three parts: (1) modifications to the SLUB allocator [110], (2) shadowing global objects, and (3) analysis and instrumentation to utilize the above runtime support.

SLUB Allocator. In our target kernel, most distinguishing regions are allocated from the SLUB allocator. There are two general types of slabs, i.e., named ones and unnamed ones. Named slabs are usually dedicated to a single data structure, if not merged, while the

unnamed ones are for `kmalloc`. Our implementation for shadow objects follows the same design philosophy: we make read access as efficient as possible at the expense of increasing the cost for write operations. Specifically, when SLUB allocates page(s) for a new slab, we allocate a shadow slab of the same size and map it as read-only at a fixed offset (4GB) from the original slab. By doing so, shadow objects can be statically located by adding this fixed offset. Similar to `kmemcheck`, we added one more field in the `page` structure to record the PFN of the corresponding shadow page. Writing to shadow objects requires an additional operation: given a pointer to a shadow object, we (1) subtract the fixed offset to locate the page for its corresponding normal object; (2) retrieve the `page` structure of the normal object and find the PFN for its shadow page; and (3) calculate the VA for the shadow object in the shadow address space, performs a context switch, and write to the VA.

Because recent Linux kernel merges slabs with similar allocation size and compatible flags, we also added one additional flag for `kmem_cache_create` to prevent slabs used for distinguishing regions from merging with other slabs. Finally, since `kmem_caches` for `kmalloc` are created during initialization, we modified this procedure to create additional caches for distinguishing regions allocated through `kmalloc`. Finally, because some slab metadata are now security-critical, we manually included them as distinguishing regions.

Global Objects. While most distinguishing regions are dynamically allocated, some of them are statically allocated in the form of global objects, such as `init_task`. Shadow objects for global objects are allocated during kernel initialization. In particular, we allocate shadow memory for the entire `.data` section, copy all the contents to populate the shadow memory and then map it in the same way as described above, so that we can use a uniformed procedure to access both heap and global objects.

Analysis and Instrumentation. Since distinguishing regions contain thousands of data structures, we use automated instrumentation to instruct the kernel to allocate and access shadow objects. Specifically, we first identify all allocation sites for objects in distinguishing regions and modify the allocation flag. If the object is directly allocated via `kmem_cache_alloc`, we will also locate the corresponding `kmem_cache_create` call and modify the creation flag. Next, we identify all pointer arithmetic operations for accessing distinguishing regions

and modify them to access the shadow objects (for both read and write access). Finally, we identify all write accesses to distinguishing regions, including `memcpy` and `memset`, and modify them to invoke our atomic operations instead. Our analysis and instrumentation do not automatically handle inline-assembly; fortunately inline-assembly is rare for the AArch64 kernel. There are only 299 unique assembly code snippets, most of which are for accessing system registers, performance counters, and atomic variables. Besides, only a few distinguishing regions are accessed by inlined-assembly, so we handle them manually in a case-by-case manner.

4.7.4 Kernel Stack Randomization

Since we use virtual address space for data-flow isolation, performing a context switch for every stack write is not feasible. Thus, we used a randomization-based approach to protect the stack. However, all randomization-based approaches must face two major threats: lack of entropy and information disclosure [76]. We address the entropy problem by mapping kernel stack to a unused VA above the logical map (top 256GB). Because kernel stacks are small (16KB), we have around 24-bit[2] of entropy available.

We contain the risk of information leak as follows. First, when performing safe stack analysis, we mark functions like `copy_to_user` as unsafe, so as to prevent stack addresses from leaking to user space. The safe stack also eliminates stack address leaked to kernel heap, so even with the capability of arbitrary memory read, attackers would not be able to pinpoint the location of the stack. As a result, there are a few special places that can be used to locate the stack, such as the `stack` pointer in the `task_struct` and the page table. To handle the formal case, we store the real stack pointer in a redirection table and replace the original pointer with an index into the table. To protect this table, we map it as inaccessible under normal context. Similarly, to prevent attackers from traversing page tables through arbitrary memory read, page tables for the shadow stacks (i.e., top 256GB) are also mapped as inaccessible under normal context. Accessing these protected tables is similar to writing distinguishing regions, which disables interrupt, performs a context switch, finishes the

[2] 38-bit (256GB) for unused space above kernel, minus 14-bit size.

operation, and restores the context and interrupt. Please note that because we un-map memory pages used for kernel stack from their original VA in logical mapping, attackers can acquire the PFN of the memory by checking un-mapped page table entries. However, this does not reveal the randomized VA of the re-mapped stack and thus cannot be used to launch attacks. Finally, the randomized stack addresses may be leakable through TLB-based side channel attacks [91]. Launching this attack requires the TLB implementation to create a TLB entry on access permission error but not on translation error. For ARMv8 VMSA, whether the TBL will create an entry on access permission error is vendor-specific. Based on experiment result, on our testbed (NVidia Denver), there is no observable performance difference between translation page fault and access permission page fault, so the TLB-based side channel is not feasible. Although such attack may be feasible on other SoC chips, as briefly discussed in [91], such attack can also be defeated via introducing random noise in page fault handling.

4.8 Evaluation

To evaluate our prototype, we designed and performed experiments in order to answer the following questions:

- How precise is the region-inference algorithm InferDists (§4.8.2)?
- How effective is our protection mechanism, ProtectDists, in blocking unauthorized attempts to access distinguishing regions through memory corruption (§4.8.3)?
- How much overhead is incurred by our protection mechanism (§4.8.4)?

4.8.1 Experimental setup

We use Google Nexus 9 as our testing device, which embeds a duo-core ARMv8 SoC and 2GB memory. We retrieved the kernel source code from the Android Open Source Project's repository (flounder branch, commit lollipop-release), and applied patches from the LLVMLinux project [195] to make it compatible with LLVM toolchain (r226004). Besides these patches, we modified 64 files and around 1900 LoC of the kernel.

All of our analysis passes are based on the iterative framework from KINT [208], with our own call graph and taint analysis. We found this framework to be more efficient than processing a single linked IR. Our point-to analysis is based on [39], which we extended to be context-sensitive with the technique proposed in [31], and ported to the iterative framework. The total LoC for analysis, excluding changes to the point-to analysis, is around 4400. And our instrumentation pass includes around 500 LoC.

4.8.2 Distinguishing Regions Discovery

Control data. For the Nexus 9 kernel, our analysis identified 6192 code pointers. Among them, 991 are function arguments and 11 are return values. With safe stack protection, these pointers do not require additional instrumentation. For the rest of the code pointers, 1490 are global variables and 3699 are fields over 783 data structures.

Non-Control data. The error codes we used were `EPERM`, `EACCES`, and `EROFS`. Overall, our analysis identified 526 functions as capable of returning permission-related errors; 1077 function arguments, 279 global variables and 1731 data fields over 855 data structures as security-critical.

Next, we measure the accuracy of our analysis. For measuring false positives, we manually verified if the reported data regions are actually involved in the access control decision. Among the 1731 data fields, our manual verification identified 491 fields over 221 data structures as not sensitive, so the empirical false positive rate is about 28.37%. However, most of the false positives (430 data fields over 196 data structures) are introduced by one single check in function `dbg_set_powergate`, which invokes a generic power management function `rpm_resume`; this generic function in turn, invokes power management functions from different drivers through callback function pointers. Since our call graph analysis is context-insensitive, this resulted in including many power management related data structures. If we blacklist this particular check, the false positive rate is only 3.52%.

For false negatives, since our analysis is sound on inferring distinguishing regions, there should be no false negatives. However, because the effectiveness of our approach depends on the correctness of our assumptions (see §4.9), we still wanted to check if all well-known

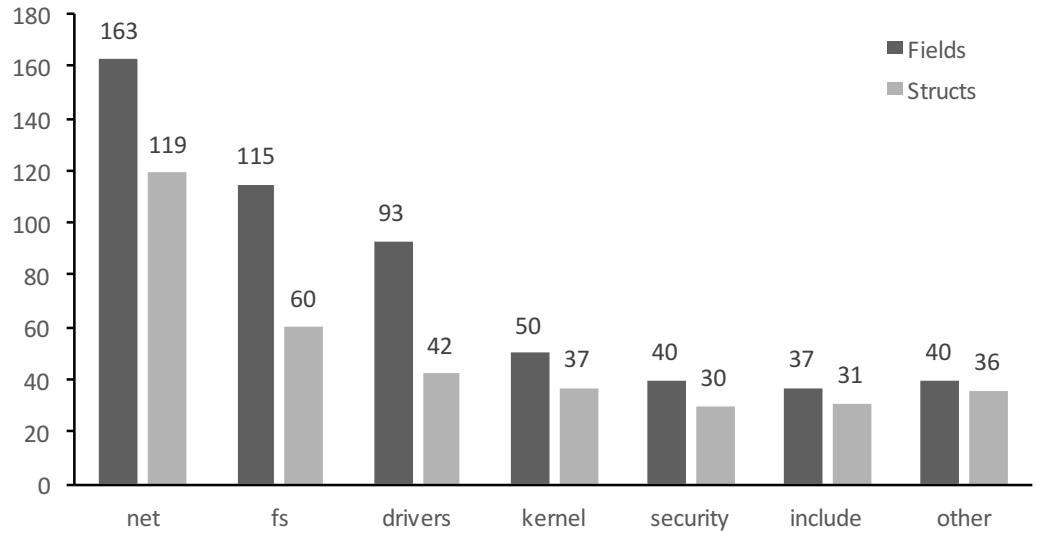

Figure 9: Data fields categorized by their usage.

sensitive data structures like the `cred` structure are included. The manual verification showed that all the well-known data structures like `av_decision`, `cred`, `dentry`, `file`, `iattr`, `inode`, `kernel_cap_struct`, `policydb`, `posix_acl`, `rlimit`, `socket`, `super_block`, `task_struct`, `thread_info`, `vfsmount`, `vm_area_struct` were included. Because of page limitations, we omit the detailed list of the discovered structures in this paper and provide them as part of an extended technique report. Here, we provide some high level statistic instead. Figure 9 shows the categories of discovered data field according to where they are used for access control[3]. The top 3 sources of distinguishing regions are network (mainly introduced by netfilter), file system, drivers and core kernel.

Sensitive pointers. Combining both control and non-control inference results, we have a total of 4906 data fields over 1316 data structures as the input for sensitive pointer inference. This step further introduced 4002 fields over 1103 structures as distinguishing regions. So, for the target kernel, KENALI should protect 2419 structures, which is about 27.30% of all kernel data structures (8861).

[3]We categorized them based on use because most of the structures are defined in header files that are not well categorized.

Table 6: Effectiveness of KENALI against different exploit techniques.

	ret2usr	cred	SELinux	RO Mount
Stock	✓	✗	✗	✗
KENALI	✓	✓	✓	✓

4.8.3 Security Evaluation

In this subsection, we first discuss the potential false negatives introduced by point-to analysis; then we use concrete attacks to show the effectiveness of our protection.

Theoretical limitation. Although our analysis is sound, because the point-to analysis is not complete (which is a typical problem for all defense mechanisms that rely on point-to analysis, including CFI, DFI and WIT), we may allow write operations that should never write to distinguishing regions to overwrite those critical data. To reduce the potential false negatives introduced by point-to analysis, we try to improve its precision by making the analysis field-sensitive and context-sensitive. Here, we provide an empirical estimation of this attack surface by measuring how many allocation sites can a pointer points to. The results showed that the majority of pointers (97%) can only point to one allocation site.

Real attacks. Since we could not find a real-world attack against our target device, we back-ported CVE-2013-6282 to our target kernel and attacked the kernel with techniques we discussed in §4.4. As shown in Table 6, KENALI was able to stop all attacks.

4.8.4 Performance Evaluation

In this subsection, we evaluate the performance overhead introduced by KENALI from the following perspectives: (1) instrumentation statistics; (2) overhead for our atomic operations and core kernel services; (3) overhead for user-mode programs; and (4) memory overhead incurred by shadow objects.

Instrumentation overhead. For instrumentation overhead, we report the following numbers.

Reallocated stack objects. Recall that to implement safe stack, we relocate unsafe stack objects to the heap. Since heap allocation is more expensive, we want to measure how many objects are relocated. Among the 155,663 functions we analyzed, there are 26,945

stack allocations. Our inter-procedural analysis marked 1813 allocations (6.73%) across 1676 functions as unsafe. Among these 1676 unsafe functions, there are 170 unsafe stores (e.g., storing a stack pointer to the heap) and 308 potential unsafe pointer arithmetic. The rest of them are due to indirect calls where the target functions cannot be statically determined (i.e., function pointers that are never defined), so we conservatively treat them as unsafe. As a comparison, the original safe stack analysis used in [108] marked 11,528 allocations (42.78%) across 7285 functions as unsafe.

Allocation sites. Data structures in the distinguishing regions can be allocated in four general ways: as global objects, from heap, on stack, or as an embedded object of a larger structure. Our analysis handles all four cases, with one limitation: our prototype implementation only handles heap objects allocated from SLAB. Overall, for the 2419 input structures, we were able to identify allocation sites for 2146 structures and cannot identify allocation sites for 385 structures. Note that the total number is larger than the input because the result also (recursively) included parent structures of embedded structures. We manually analyzed the result and found that some of those missing data structures like `v4l2_ctrl_config` actually are never allocated in our target kernel configuration, while others are allocated directly from page allocator, such as `f2fs_node`, or casted from a disk block, such as `ext4_inode`.

Instrumented instructions. For the target kernel, our analysis processed a total of 158,082 functions and 619,357 write operations. Among these write operations, 26,645 (4.30%) were identified as may access distinguishing regions whose allocation sites were successfully located and thus were replaced with atomic write primitives. Within instrumented write operations, only two operations were marked as unsafe and need to be instrumented to guarantee write integrity. Besides, we also instrumented 137 `memcpy`/`memset` calls.

Binary size. In this experiment, we measure the binary size increment introduced by KENALI. The result is shown in Table 7. As we can see, Clang-generated binaries are smaller than GCC (version 4.9.x-google 20140827), and the binary size increase is minor, so the instrumented binary is only slightly larger than the stock GCC-compiled kernel.

Micro benchmarks. For micro benchmarks, we measured two targets: (1) the overhead

Table 7: Compressed kernel binary size increment.

	Stock	Clang	KENALI	Increase
Size (in bytes)	7,252,356	6,796,657	7,165,173	5.42%

for a context switch and (2) the overhead for core kernel services.

Context switch. In this experiment, we used the ARM cycle count (`PMCCNTR_EL0`) to measure the latency for a round-trip context switch under our protection scheme. For each round of testing, we performed 1 million context switches, and the final result is based on five rounds of testing. The result, with little deviation, is around 1700 cycles. Unfortunately, lacking access to hypervisor mode and secure world on the target device, we cannot directly compare the expense for context switching to hypervisor and the TrustZone on that device. A recent study [151] showed that on an official Cortex-A53 processor, minimal round-trip cost for a guest-host switch is around 1400 cycles, and around 3700 cycles for a non-secure-secure switch (without cache and TLB flush).

LMBench. We used LMBench [127] to measure the latency of various system calls. To measure how different techniques affect these core system services, we consider four configurations: (1) baseline: unmodified kernel compiled with clang; (2) CI: kernel with only life-time kernel code integrity protection (§4.7.2); (3) Stack: code integrity plus stack protection (§4.7.4; and (4) KENALI: full DFI protection. The result, averaged over 10 times, is shown in Table 8 [4]. For comparison, we also included numbers from Nested Kernel [62], which also provides lifetime kernel code integrity; KCoFI [56], which enforces CFI over the whole kernel; and SVA [57], which guarantees full spatial memory safety for the whole kernel. Please note that because these three systems are evaluated on x86 processors and with different kernels, the comparison can only be used as a rough estimation.

As we can see, for syscalls that involve distinguishing regions manipulation, KENALI tends to have higher performance overhead than KoCFI but lower than SVA. But for syscalls that do not involve distinguishing regions manipulation, e.g., null syscall, KENALI

[4]Because the number for Nested Kernel (PerspicuOS) was reported in a graph, the numbers here are estimations. Because the original SVA paper did not use LMBench, we used the number reported in the KCoFI paper.

Table 8: LMBench results.

Benchmark	CI	Stack	KENALI	NK [62]	KCoFI [56]	SVA [57]
null syscall	0.99x	1.00x	1.00x	∼1.0x	2.42x	2.31x
open/close	0.97x	0.99x	2.76x	∼1.0x	2.47x	11.00x
select	1.00x	1.05x	1.42x	-	1.56x	8.81x
signal install	1.34x	1.32x	1.30x	∼1.0x	2.14x	5.74x
signal catch	0.99x	1.11x	2.23x	∼1.0x	0.92x	5.34x
pipe	0.95x	1.02x	3.13x	-	2.07x	13.10x
fork+exit	1.31x	1.40x	2.18x	∼2.9x	3.53x	-
fork+execv	1.50x	1.55x	2.26x	∼2.5x	3.15x	-
page fault	1.61x	1.69x	1.71x	∼1.0x	1.11x	-
mmap	1.60x	1.66x	1.63x	∼2.5x	3.29x	-

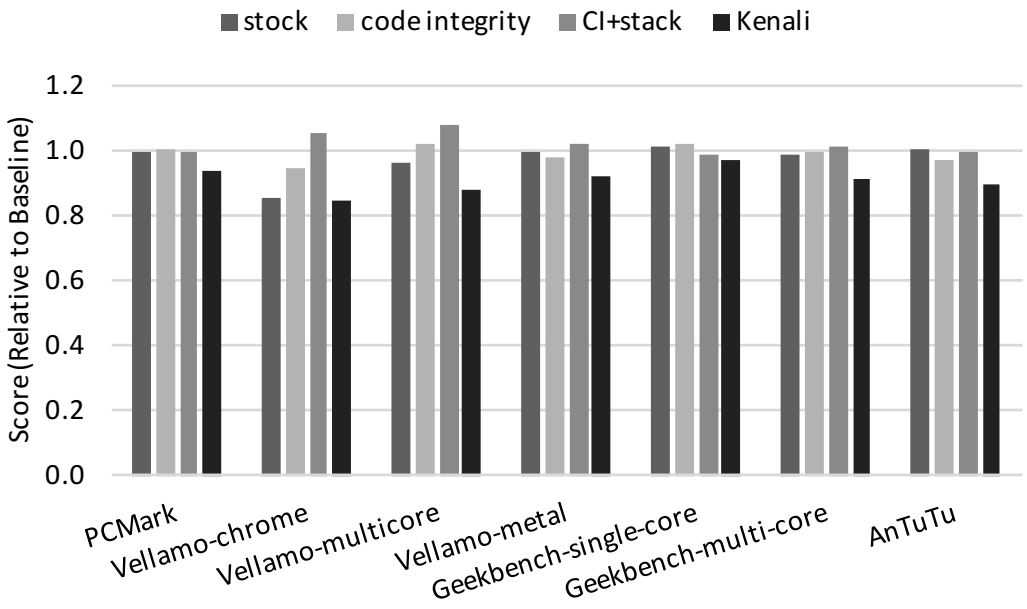

Figure 10: Benchmark results from four standard Android benchmarks.

has no observable performance overhead. The overhead for enforcing lifetime code integrity is similar to PerspicuOS.

Android benchmarks. To measure the performance impact on user-mode programs, we used four standard Android benchmarks: AnTuTu, Geekbench, PCMark, and Vellamo. All of these benchmarks simulate typical real-world scenarios, including web browsing, video playback, photo editing, gaming, etc. The configurations we used are similar to LMBench. The result is shown in Figure 10. As we can see, with KENALI's protection, the slowdown for these user-mode benchmarks is between 7% - 15%, which we think is acceptable.

Memory overhead. To measure the memory overhead introduced by KENALI (due to

Table 9: Number of kmem_cache with shadow objects and the number of pages used by shadow objects.

	# kmem_cache	# pages	MB	% of total slab	% of total memory
Reboot	85	9945	38.85	65.11%	1.90%
Bench	85	9907	38.70	59.79%	1.89%

the use of shadow objects), we modified the `/proc/slabinfo` interface to report slabs with shadow objects. Based on this, we calculate how many additional pages are allocated and their percentage to the whole memory pages used by all slabs. We acquire this number at two time points: (1) after fresh reboot and (2) after finishing the AnTuTu benchmark. The result is shown in Table 9.

4.9 Limitations and Future Work

In this section, we discuss limitations of our current design and implementation, insights we learned, and possible future directions.

4.9.1 Cross-platform

Although we choose AArch64 for the prototype, core techniques of Kenali are generic to most other commodity platforms. Specifically, data-flow isolation can also be implemented with the help of segmentation on the x86-32 architecture [223], WP-bit on the x86-64 architecture [62], and access domain on the AArch32 architecture [235]. For shadow objects, our current design is based on SLAB allocator, which is used by many *nix kernels like Solaris and FreeBSD. In theory, it can also be implemented on any memory pool/object cache based allocator. The rest two techniques, WIT and safe stack, are both platform-independent.

4.9.2 Better architecture support

A majority of Kenali overhead can be eliminated by having better hardware support. For example, with the application data integrity (ADI) feature from the SPARC M7 processor [148], (1) there would be no context switch for accessing distinguishing regions, and (2) all memory overhead introduced by shadow objects can be eliminated. With the kernel guard technology from Intel [92], enforcing lifetime kernel code integrity can be less expensive. We explored this direction in §5.

4.9.3 Reliability of assumptions

Our static analysis, InferDists, relies on two assumptions: (1) there is no logic bug in the access control logic, and (2) there is no semantic error (i.e., failure of access control checks should always leads to returning corresponding error codes). For most cases, these assumptions usually hold, but may sometimes be violated [13, 38, 130]. However, we believe Kenali still makes valuable contributions, as it is an automated technique that can provide a strong security guarantee against memory-corruption-based exploits. In other words, by blocking exploits against low-level vulnerabilities, future research could focus on eliminating high-level bugs such as logical and semantic bugs.

4.9.4 Use-after-free

Our current design of Kenali focuses on spatial memory corruptions; thus it is still vulnerable to temporal memory corruptions, such as use-after-free (UAF). For example, if the `cred` of a thread is incorrectly freed and later allocated to a root thread, the previous thread would acquire the root privilege. However, Kenali still increases the difficulty of exploiting UAF vulnerabilities: if the wrongly freed object is not in distinguishing regions, exploiting such vulnerabilities cannot be used to compromise distinguishing regions. At the same time, many distinguishing regions data structures like `cred` already utilize reference counter, which can mitigate UAF. So we leave UAF mitigation as future work.

4.9.5 DMA protection

Since DMA can directly write to any physical address, we must also prevent attackers from using DMA to tamper distinguishing regions. Although we have not implemented this feature in Kenali yet, many previous works [173] have demonstrated how to leverage IOMMU to achieve this goal. Since IOMMU is also available on most commodity hardware, we expect no additional technical challenges but only engineering efforts.

4.10 Summary

In this chapter, we presented Kenali, a principled and practical approach to defeat all memory-corruption-based kernel privilege escalation attacks. By enforcing important kernel

security invariants instead of individual exploit techniques, KENALI can fundamentally prevent all attacks. And by leveraging novel optimization techniques, KENALI only imposes moderate performance overhead: our prototype implementation for an Android device only causes 5-17% overhead for typical user-mode benchmarks.

CHAPTER V

IMPROVE SECURITY AND PERFORMANCE WITH HARDWARE-ASSISTED DATA-FLOW ISOLATION

5.1 Motivation

Memory corruption vulnerabilities are the root cause of many modern attacks. To defeat such attacks, many security features have been commoditized, including NX-bit (No-eXecute), Supervisor Mode Execution Protection (SMEP), Supervisor Mode Access Prevention (SMAP), Memory Protection Extension (MPX), which have provided a strong foundation for security in today's computer systems. However, while these hardware-based security features are very efficient, they do not provide adequate protection against modern, complex memory-corruption-based attacks. For example, NX-bit can eliminate simple forms of code injection attacks, but cannot stop code-reuse attacks such as return-to-libc attack [68], return-oriented programming (ROP) [176], COOP [167], and non-control data attacks [41, 88, 182].

To defeat these new attacks, researchers continue to develop new hardware-based mechanisms. For example, hardware-based shadow stacks have been proposed to protect return addresses from tampering by adversaries [218, 114, 150]. Hardware-based control-flow integrity (CFI) has also been proposed to prevent code-reuse attacks, with various trade-offs [55, 103, 64, 63]. Furthermore, a number of other approaches have been proposed to eliminate the root cause of these memory corruption vulnerabilities [69, 133, 134, 212].

In §4, we demonstrated how program analysis and lightweight isolation can be combined together to provide more efficient protection against memory corruption based exploits. Our key observation is that even with hardware support, enforcing memory safety for the whole application is still too expensive for practical use, e.g., WatchDogLite [134] imposes 29% slowdown on SPEC CINT 2006 benchmarks. To further reduce performance overhead, one promising direction is to divide the memory into different regions—one for sensitive data (e.g., function pointers) and the other for the rest (e.g., application data). Then, we

Table 10: Comparison between HDFI and other isolation mechanisms.

	Mechanism	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Software-based	Randomization [170, 108]	Y	N	N	Y	Y	Y	low
	Masking [170, 108, 56]	Y	N	N	Y	Y	N	moderate
	Access control list [74, 35]	N	N	Y	Y	Y	N	high
Hardware-based	x86 memory segment [223, 108]	Y	N	N	N	Y	N	low
	ARM access domain [235]	Y	Y	N	N	Y	N	moderate
	Virtual address space [182]	Y	Y	Y	Y	Y	N	high
	Privilege level	Y	Y	Y	Y	N	N	moderate
	Virtualization [173]	Y	Y	Y	Y	N	N	high
	TrustZone [16]	Y	Y	Y	Y	N	N	very high
	ADI [148]	N	N	Y	Y	Y	N	low
	HDFI	N	N	N	Y	Y	N	low

enforce memory safety only over the sensitive region [108, 170, 182]. There are two major advantages of this approach. First, sensitive data is usually a smaller set than normal data, and less data implies fewer checks and less performance overhead. Second, the safety of memory operations over sensitive data is easier for static verification. For example, because pushing/popping data onto/from stack is always safe, once we isolate the stack slots used to store return addresses, we can guarantee memory safety for return addresses without any runtime check. With these two advantages, we can significantly reduce the number of runtime checks, thereby making memory safety more affordable. However, implementing this strategy on commodity hardware is non-trivial due to the lack of *an efficient, fine-grained mechanism for data isolation.*

Table 10 compares existing software-based (top half) and hardware-based (bottom half) isolation mechanisms on *commodity hardware* over seven criteria: **(1)** whether data shadowing is required, **(2)** whether context switch is required for data access, **(3)** whether liveness tracking is required, **(4)** is available on 64-bit mode, **(5)** whether they can be used for self-protection, **(6)** is vulnerable to information leak, and **(7)** performance overhead. Self-protection means whether the mechanism can be used to prevent attacks from the same privilege-level. Performance overhead is measured by comparing one instrumented read/write operation against a normal memory read/write operation. And as there is no public benchmark result for ADI, so this conclusion is purely based on their presentation [148].

The most apparent problem is that the two most efficient hardware-based mechanisms—segment in x86 and access domain in ARM processors, are absent on 64-bit mode. As

a result, security solutions in modern processors must make a trade-off between security and performance—solutions that opt for performance (e.g., by using randomization based protection) are usually subject to information disclosure or brute-force based attacks [76, 51]; while solutions that opt for security (e.g., by leveraging context switch or masking) usually yield poorer performance [170, 56, 182].

Moreover, even if we managed to bring back the segment and access domain, these mechanisms are still inadequate. Specifically, because they are all coarse-grained, if we want to isolate data at a smaller granularity (e.g., function pointers) and preserve a program's original memory layout, then we must perform *data shadowing*. Unfortunately, data shadowing breaks data locality and requires extra steps to retrieve the shadow data. This introduces additional performance overhead [60]. Furthermore, data shadowing also introduces unavoidable memory overhead.

To overcome these limitations, we propose *hardware-assisted data-flow isolation* (HDFI), a new fine-grained data isolation mechanism. To eliminate data shadowing, HDFI enforces isolation at machine word granularity by virtually extending each memory unit with an additional tag. HDFI's tags are associated with memory units' *physical addresses*, so attackers cannot tamper or bypass the protection by mapping the same physical page to different virtual addresses. Moreover, instead of using static partition, the tag is defined by data-flow. Inspired by the idea of data-flow integrity [34], HDFI defines the tag of a memory unit by the last instruction that writes to this memory location; then at memory read, it allows a program to check if the tag matches what is expected. This capability allows developers to enforce different security models. For example, to protect the *integrity* of sensitive data, we can enforce the Biba Integrity Model [24]. In particular, we can use the tag to indicate integrity level (IL) of the corresponding data: sensitive data has `IL1` and normal data has `IL0`. Next, we assign IL to write operations based on the data-flow. That is, we use static analysis to identify write operations that can manipulate sensitive data, and allow them to set the memory tag to `IL1`; all other write operations will assign to the tag to `IL0`. Finally, when loading sensitive data from memory, we check if the tag is `IL1` (see §5.3 for a concrete example). HDFI can also be used to enforce *confidentiality*, i.e., the Bell–LaPadula

Model [21]. For instance, to protect sensitive data like encryption keys, we can set their tag to `SL1` (secret level 1), and enforce that all untrusted read operations (e.g., when copy data to an output buffer) can only read data with tag `SL0`.

5.2 Threat Model and Assumptions

In this work, we focus on preventing memory corruption based attacks; therefore we follow the typical threat model of most related work. That is, we assume that software may contain one or more memory vulnerabilities that, once triggered would allow attackers to perform arbitrary memory reads and writes. We do not limit what attackers would do with this capability, as there are many different attack vectors given this capability. As a hardware-based solution, we also do not limit where the vulnerabilities are: they can be in user-mode applications, OS kernel, hypervisor, etc. However, we assume all hardware components are trusted and bug free, so attacks that exploit hardware vulnerabilities, such as the row hammer attack [106], are out-of-scope.

Similar to NX-bit, HDFI requires software modifications to obtain its benefits. This can be done in many ways: manual modification, compiler-based modification, static binary rewriting, dynamic binary rewriting, etc. For the example applications we demonstrated in this paper, we either manually modified the source or leveraged compiler-based approaches. However, we must emphasize that this is not a limitation of HDFI and source code is not always necessary.

5.3 Background and Related Work

This section provides the background of HDFI and compares HDFI with related work.

5.3.1 Data-flow Integrity

The goal of HDFI is to prevent attackers from exploiting memory corruption vulnerabilities to tamper/leak sensitive data. To achieve this goal, we leverage data-flow integrity (DFI) [34]. DFI ensures that the runtime data-flow cannot deviate from the data-flow graph generated from static analysis. In particular, DFI assigns an identifier to each write instruction and records the ID of the last instruction that writes to a memory position; then at each read

instruction, DFI checks whether the ID of the last writer belongs to the set allowed by static analysis. Take Example 3. This code snippet contains a buffer overflow vulnerability at line 6, which allows attackers to use `strcpy()` to overwrite the return address saved at line 3 and launch control-flow hijacking attacks. Such attacks can be prevented by checking if the return address read at line 8 is defined by the store instruction at line 3.

```
1  main:
2    add     sp,sp,-32
3    sd      ra,24(sp)
4    ld      a1,8(a1)        ; argv[1]
5    mv      a0,sp           ; char buff[16]
6    call    strcpy          ; strcpy(buff, argv[1])
7    li      a0,0
8    ld      ra,24(sp)
9    add     sp,sp,32
10   jr      ra              ; return
```

Example 3: A typical stack buffer overflow example, in RISC-V assembly.

In HDFI, we extend the ISA to perform DFI-style checks with hardware. Specifically, we leverage memory tagging to record the last writer of a memory word and provide new instructions to set and check the tag. However, instead of trying to fully replicate DFI, which would require supporting arbitrary tag size, we focus on providing isolation, i.e., using a one-bit tag to indicate the trustworthiness of the writer. Using the same example, HDFI can be utilized to prevent the attack by (1) using a new instruction `sdset1` (store and set tag) to set the tag of memory used to store return address to `1` (line 3); and (2) when loading the return address from memory for function return, using another instruction `ldchk1` (load and check tag) to check if the memory tag is still `1`. Since normal store instructions (e.g., `sd`) would set the tag to `0`, if attackers try to overwrite the return address, the `ldchk1` instruction would fail and generate a memory exception.

5.3.2 Tag-based Memory Protection

Tag-based memory protection is not new and has been explored in many previous works. For example, lowRISC [28] uses a 2-bit tag to specify if a memory address is readable and writable. Loki [229] also allows developers to specify permission with a memory address, but is more flexible, as the permission is related to the current protection domain. The problem with these approaches (including the Mondriaan protection model [216]) is that, although the

objects (memory addresses) are fine-grained, the subjects are still coarse-grained—the access permissions are applied to the whole program or the whole protection domain. However, the subjects are individual instructions in HDFI.

An alternative approach is to associate the access permission with pointers instead of memory locations. For example, Watchdog [133] and the application data integrity (ADI) [148] mechanism on SPARC M7 processors allow a program to associate memory addresses and pointers with versions (tags) and require that when accessing the memory the version of the pointer must match the version of the memory. The tricky part of this approach is how to maintain the tag of a pointer, because every pointer should have two tags: one indicating the tag of the target memory, and the other indicating the tag of the memory where the pointer is stored. Without this, attackers can still tamper with the pointers. Watchdog handles this by using shadow memory to maintain the first type of tags, but it is unclear whether or how ADI handles this issue.

Write integrity test [4] is another tag-based memory safety enforcement mechanism. It enforces that each write operation (instead of pointer) can only write to objects that are allowed by the static data-flow graph. However, since the integrity test is only enforced on write operations, WIT can only enforce data integrity, but not data confidentiality.

A common issue with all the aforementioned approaches is that they must track the liveness of memory objects, which makes the protection more complicated. For instance, in Example 3, to protect the return address, all aforementioned systems must tag the memory used for return address at prologue. Here we must pay special attention to the order of tagging and store: if store happens before tagging, the system would be vulnerable to time-of-check-to-time-of-use (TOCTTOU) attack, because the address might be modified unless the two operations are guaranteed to be atomic. Then, after the function finishes execution and returns, the current stack frame is *freed*, so the old memory position used to store the return address must be unprotected for future re-use. Here is another tricky part—if the capability system is location-based, or does not assign a new version for every memory allocation (which is very challenging for fixed tag size), then it would be subject to use-after-free (UAF) based attacks. Moreover, for software that heavily utilizes custom

memory allocators, such as browsers and OS kernels, tracking object allocation is non-trivial. Fortunately, HDFI does not need to track liveness of memory objects.

Among existing hardware features, Minos [55] and CHERI [212] are the closest to HDFI. Specifically, Minos uses one-bit tags to indicate the integrity of code pointers and updates the tag based on the Biba model [24]. CHERI [212] also uses one-bit tags to indicate whether a memory address stores a valid capability (fat pointer). This bit can only be set when the memory content is written by a capability-related instructions and is cleared when written by normal store instructions. Comparing to them, the advantage of HDFI is flexibility—as will be shown in §5.5, besides pointers, HDFI can also be used to protect generic data like `uid`; and along with the Biba model, HDFI can also be used to enforce the Bell–LaPadula model [21].

5.3.3 Tag-based Hardware

Because memory tagging is widely used for dynamic information flow tracking (DIFT), which can be very expensive when purely done in software [142]. For this reason, numerous hardware solutions have been proposed, including pure DIFT-oriented [59, 190, 55, 102], and more general, programmable metadata processing [202, 40, 67, 70]. The most significant difference of HDFI from these solutions is our emphasis on minimizing hardware changes so as to make HDFI more likely to be adopted in practice. In particular, HDFI does not require modifying register files, ALU, main memory, or the bandwidth between cache and main memory. More importantly, instead of requiring half of all physical memory dedicated to store tags (i.e., an overhead of 100%), HDFI only impose 1.56% memory overhead.

5.3.4 Memory Safety

Since memory safety issues are the root cause of many attacks [192], researchers have proposed many solutions to address this problem, including automated code transformation [137], instrumentation-based [34, 4, 135, 136], and hardware-based [69, 133, 134, 94, 212]. The biggest hurdle for adopting these solutions is their performance overhead—even with hardware assistance, the average overhead is still 29% on benchmark workloads [134]. To help further reduce the overhead, HDFI is designed to enable another optimization direction—using

isolation to limit the protection scope and only enforcing memory safety over the isolated data. Such data could be security sensitive, e.g., code pointers [108], generic pointers [54, 212], or important kernel data [182]. It could also be data that can be statically proved to be memory safe, e.g., safe stack [108]. We believe such a combination would allow us to build powerful yet efficient solutions to eliminate all memory corruption based attacks.

5.4 HDFI Architecture

In this section, we present the design of HDFI, which includes two major components: the ISA extension and the memory tagger. Our current design tags memory at machine-word granularity because most sensitive data we want to protect are of this size (e.g., pointers). For data not of this size, we can manually extend the size, or leverage compilers. To prevent attackers from creating inconsistent views of data and its corresponding tag and launching TOCTTOU attacks in a multi-core/-processor system, we require all HDFI instructions to be *atomic* (i.e., data and tag must always be loaded and stored together) and comply with the same cache consistency model as other memory accessing instructions. To avoid changing the main memory system and the data link between main memory and the processor, our current design stores all the tag information at a dedicated area called *tag table*. In our current design, tag table is allocated and initialized by the OS kernel during boot, similar to how Intel SGX reserves the secure pages (i.e., EPC pages) for enclaves [94]. Once allocated, the memory region for the tag table will be protected from malicious modification (§5.4.4).

5.4.1 ISA Extension

To enforce DFI, the authors added two high-level instructions: `SETDEF` and `CHECKDEF` [34]. Since HDFI only supports one-bit tags, in order to allow programs to use DFI-style checks to enforce the integrity/confidentiality level of memory contents, we introduce three new instructions:

- `sdset1 rs,imm(rb)`: store word and set tag to `1`.
- `ldchk0 rd,imm(rb)`: load word and check if tag equals `0`.
- `ldchk1 rd,imm(rb)`: load word and check if tag equals `1`.

Note that we do not have an instruction that explicitly sets the tag to `0` (or `sdset0` per our naming convention). Instead, HDFI implicitly sets the tag of the destination memory to `0` when written by regular store instructions, such as `sd` (i.e., store double words), `sw` (i.e., store a word), and `sb` (i.e., store a byte). However, HDFI preserves the semantics of regular load instructions, i.e., tag is not checked on regular load operations. To check the tag bit of the target memory region, HDFI provides `ldchk0` and `ldchk1`. To enable the OS kernel to capture tag mismatch, we also introduced a new memory exception, which is similar to other memory faults except for the error code.

HDFI also provides a special instruction alias `mvwtag` [1] for copying the memory from a source to a destination along with the corresponding tag bits. This special operation is necessary to achieve optimal performance in modern system software. Specifically, modern OS kernels like Linux use copy-on-write (CoW) to share memory between the parent process and its child processes. However, if we use normal `sd` operations to perform the copy, it could break HDFI-protected applications because the tag information is lost; on the other hand, we also cannot use `sdset1` because it allows attackers to abuse this feature to tag arbitrary data. To solve this problem, we introduced the `mvwtag` instruction to allow OS kernels to copy data while preserving the tag. Please also note that because `memcpy` can cause memory corruption, we do not recommend using `mvwtag` to implement `memcpy` unless the developer can guarantee memory safety for all the invocations of `memcpy`.

5.4.2 Memory Tagger

Our hardware extension is similar to lowRISC [28]. Specifically, to simplify the implementation of the new instructions and support atomicity in a multi-core/-processor system, we modified the interface between the processor core and the cache system (including the coherence interconnect) to associate each data with its tag. In particular, when the processor core executes a memory related instruction such as `sd`, `sdset1` or `ld`, it sends a request to the data cache(s). This request includes a data field and a command field. HDFI adds one tag bit to the data field, so for every memory write request, data is always stored with the

[1] Since we do not extend general register files with tag, this operation is an alias for two instructions: load data and tag from source into a special register then store them to the destination.

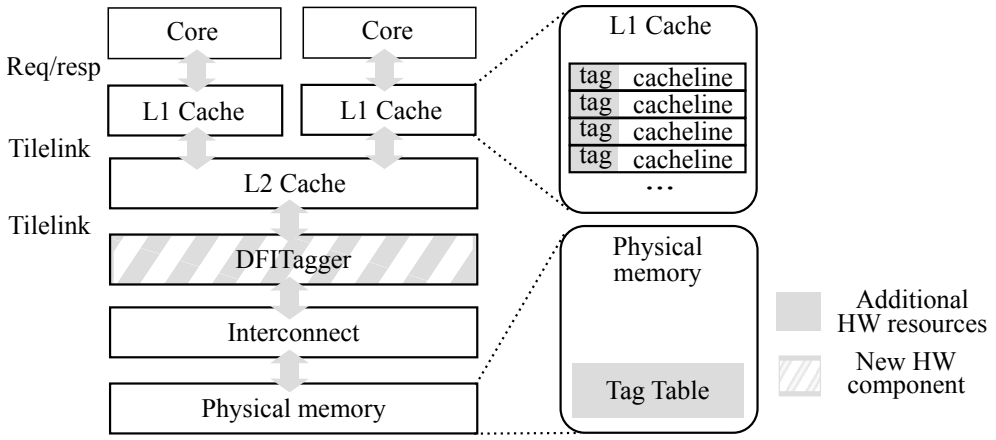

Figure 11: Design of HDFI.

tag; and for memory load requests, tags can be (not always, see §5.4.3 for detail) loaded with data.

To facilitate this, we augmented the caches to hold the tags for the cached memory units, as shown in Figure 11. To hold the tag bits for the cached memory units, the caches have a one-bit register for each machine word to store the corresponding tag. When the processor core sends a store request, the L1 cache can simply update the data and tag value with the incoming value from the core; and when the core sends a read request, the L1 data cache provides the core with the tag bit, with which the core can check whether the tag matches expected value or not.

While the L2 cache can also be augmented similarly to hold the tags for each memory unit, we believe it is not feasible to add the tag bits physically to the external main memory. For this reason, we added an additional module DFITAGGER in between the L2 cache and the main memory, which decomposes memory accesses from the L2 cache to separate data accesses and tag accesses. Data accesses are handled as usual and tag accesses are handled as follows. HDFI preserves a memory chunk to be used as tag table (Figure 11), which acts as a huge bit vector to store tag bits. When the L2 cache issues a memory access, DFITAGGER maps the physical address to a table entry of the tag table and generates a tag access.

5.4.3 Optimizations

Unfortunately, the additional memory accesses to the tag table introduce non-negligible performance overhead. More specifically, without any optimization, HDFI will double the memory accesses because for every cache miss, DFITAGGER needs to issue one data access and another tag access. To minimize this impact, we developed several optimization techniques.

5.4.3.1 Tag Cache

The most straightforward way of reducing the overhead is caching, so we introduced a tag cache within the DFITAGGER to exploit the locality of memory accesses. Moreover, tag cache also allows DFITAGGER to fetch a set of tags from the main memory in the cache line granularity to reuse the existing memory interface. For example, a cache line in the Rocket Core is 64 bytes. To handle one cache miss, DFITAGGER only needs 8 tag bits (one bit per eight bytes), but for the fixed size of memory interface, it has to fetch 64 bytes from the tag table. In fact, this 64-byte unit, which we call one *tag table entry*, naturally stores the tags for a 4 KB memory block; so tag cache allows us to generate only one memory access per 4 KB data access.

5.4.3.2 Tag Valid Bits

The second optimization technique takes advantage of the fact that most of the memory loads are *not* checked, so there is no need to always refill the cache line with corresponding tag bits. Leveraging this observation, we add a *Tag Valid Bit* (TVB) to each memory unit in the caches to further reduce unnecessary accesses to the tag table. TVB is updated as follows. When the cache has to refill a line but the request from the inner cache or the processor core does not explicitly asks for tag bits, the cache generates a refill request to the outer cache or DFITAGGER, and clears the TVB for the memory units in the line. Later, if an incoming load (with tag) request hits in the cache, but the TVB for the corresponding memory unit is not set, the cache will refill the line again with the valid tags. Note that any write hit will set the TVB because store operations always update the tag bit. Finally, when a cache line is evicted and written back to main memory, the cache forwards TVB to

DfiTagger, so the later can update the tag cache accordingly.

5.4.3.3 Meta Tag Table

The third technique leverages the fact that most of the memory units are tagged with `0` and only a few ones will be tagged with `1`. This means that most tag table entries would be filled with `0`. To take advantage of this observation, DfiTagger maintains a *Meta Tag Table* (MTT) in the main memory and a *Meta Tag Directory* (MTD) as a register. Each bit of the MTT entries is set to `1` if the corresponding tag table entry contains `1`, and each bit of MTD is set to `1` if the corresponding MTT entry has `1`. Utilizing them, DfiTagger can avoid fetching tag table entries from the main memory if they are filled with `0`. It also enables DfiTagger to avoid (1) updating the tag table entry for a given write miss if that entry is filled with `0`; and (2) write back to main memory if both the evicted tag cache and the main memory copy are filled with `0`.

5.4.4 Protecting the Tag Tables

The design of Hdfi requires that the tag table and the meta tag table in the main memory are protected from the malicious modifications. To do so, we leverage the fact that DfiTagger is sitting between the cache and the main memory, hence we can use it to mediate all modifications to the main memory. That is, once the memory chunk used for tag tables are assigned to DfiTagger, it drops any access to this memory chunk. Because tag is always provided by DfiTagger, this effectively prevents any malicious modifications to the tag tables. Note that our current design cannot prevent DMA-based attacks; we will discuss this issue in §5.9.

5.5 Security Applications

In this section, we demonstrate how Hdfi can be utilized to build security solutions with simplified designs, improved performance, and better security. We want to use these examples to highlight the generality of Hdfi (i.e., the ability to support different security applications), as well as its ease of adoption. Regarding backward compatibility, it completely depends on the security solution. Some security mechanisms like shadow stack could allow mixing

protected and unprotected code, but other solutions like VTable protection will not allow such mixing.

In each application example, we focus on protecting one type of security critical data, such as return addresses, function pointers, etc. However, as there is no overlapping between the protected data (i.e., the meaning of the tag bit is not ambiguous), we can integrate all mechanisms together to maximize the defense against memory corruption based attacks.

To implement these examples, we either directly modified the source code or augmented compilers to emit HDFI's new instructions. However, we want to emphasize again that this is not a limitation of HDFI—as long as a security solution can make the target program use HDFI's new instructions, it will be able to leverage the isolation provided by HDFI.

5.5.1 Shadow Stack

In Example 3, we have demonstrated how to use HDFI to implement a virtual shadow stack for protecting the return addresses. To implement this scheme, we just need to change 6 lines in GCC (Example 4). Implementation in the LLVM toolchain is similarly simple, with only 4 lines of changes—in function `storeRegToStackSlot`/`loadRegFromStackSlot`, which are invoked at function prologues/epilogues, we use `sdset1`/`ldchk1` instead of normal store/load. Because these functions are also used to handle register spills/restores, our (LLVM-based) shadow stack also protects spilled registers, which can also be an attack vector [51].

```
1  char *riscv_output_move (rtx dest, rtx src) {
2      // if dest == REG && src == MEM
3      if (flag_safe_stack && (REGNO (dest) == RETURN_ADDR_REGNUM))
4          return "ldchk1\t%0,%1";
5      else
6          return "ld\t%0,%1";
7      // if dest == MEM && src == REG
8      if (flag_safe_stack && (REGNO (src) == RETURN_ADDR_REGNUM))
9          return "sdset1\t%z1,%0";
10     else
11         return "sd\t%z1,%0";
12 }
```

Example 4: How to use HDFI to implement shadow stack in GCC, with only 6 lines of changes.

Supporting context saving and restoring like `setjmp/longjmp` has always been a challenge for hardware-based shadow stacks [218, 114, 150]. However, for a HDFI-based shadow stack, supporting this feature is straightforward—just like saving registers to the stack,

when saving current context to `jmp_buf`, we set the tag of the corresponding memory to `1`. Then, when restoring the context, we check if the memory tag is still `1`. If attackers try to overwrite `jmp_buf`, the load check will fail. Furthermore, because HDFI-based shadow stack is still memory-based, it naturally supports deep recursion. It can even support modifying return addresses as long as they are always stored using `sdset1` and loaded with `ldchk1`. Finally, unlike SmashGuard [150], because HDFI is orthogonal to the execution privilege level, HDFI-based shadow stack does not need any support from the OS kernel and can also be used to protect kernel stacks.

5.5.2 Standard Library Enhancement

Runtime libraries like the dynamic linker (`ld.so`) and the standard C library are important parts of every program's runtime security. Unfortunately, many compiler-based security solutions neglected them, thus leave holes for attacks [101, 164, 30]. In this subsection, we describe enhancements made to the libraries to prevent attacks.

Heap Metadata Protection. Many standard C libraries like `glibc` (GNU C Library) uses a variant of Doug Lea's Malloc [112] that supports multi-threading, called (`ptmalloc`). `ptmalloc` uses double-linked lists to manage freed memory chunks. When removing a memory chunk from this list, it performs a general unlinking process (Example 5). If there exist a heap buffer overflow vulnerability, attackers can exploit this vulnerability to tamper with these metadata (pointers), which will allow attackers to overwrite an arbitrary address with arbitrary data [101]. Moreover, despite that many integrity checks have been applied to the heap implementation to stop heap-based attacks, attackers still find their ways to bypass them [75, 78].

To prevent such attacks, we can leverage HDFI to protect the integrity of these metadata—similar to return addresses, when linking a freed chunk, we set the tags of forward and backward pointers to `1`; then when unlinking a chunk, we check if the tag is still `1`. By doing so, if attackers overwrite these pointers (with normal writes), the tag will be set to `0`, which will be captured by the load check.

Global Offset Table Protection. Global Offset Table (GOT) is a data structure for

```
void unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr (check_action, "corrupted double-linked list", P);
  else {
    FD->bk = BK;
    BK->fd = FD;
    if (!in_smallbin_range (P->size)
        && __builtin_expect (P->fd_nextsize != NULL, 0)) {
      assert (P->fd_nextsize->bk_nextsize == P);
      assert (P->bk_nextsize->fd_nextsize == P);
      if (FD->fd_nextsize == NULL) {
        if (P->fd_nextsize == P)
          FD->fd_nextsize = FD->bk_nextsize = FD;
        else {
          FD->fd_nextsize = P->fd_nextsize;
          FD->bk_nextsize = P->bk_nextsize;
          P->fd_nextsize->bk_nextsize = FD;
          P->bk_nextsize->fd_nextsize = FD;
        }
      } else {
        P->fd_nextsize->bk_nextsize = P->bk_nextsize;
        P->bk_nextsize->fd_nextsize = P->fd_nextsize;
      }
    }
  }
}
```

Example 5: Unlinking function when a heap memory chunk is re-allocated or merged.

dynamic linking. Since GOT is modifiable by default and affects the program's control flow, GOT overwriting [164] has been used for changing control flow with memory corruption based attacks. Although GNU loader's RELocation Read-Only (RELRO) [193] provides protection for GOT, it requires to resolve all symbols at start up time of the program, leading to unavoidable performance degradation. To overcome the limitation, we implemented GOT protection enforce that whenever a dynamic linked function is invoked, the target address is loaded by `ldchk1`. To tag the initial pointer (i.e., the call to the resolver), we leveraged the fact that for position-independent executables (PIE), GOT table entries need to be patched due to ASLR; so we modified the relocation routine to tag the initial GOT values with `1`. Then during runtime, after resolving a real function address, we make the loader use `sdset1` to update the GOT value. Compared with RELRO, GOT protection keeps the advantage of lazy loading and prevents it from malicious overwriting.

Exit Handler Protection. To execute arbitrary code in the Full RELRO enabled binary which is not attackable by GOT overwriting, another attack surface is the exit handler [30]. To prevent attackers from manipulating the exit handler, pointer encryption [72] is applied

in `glibc`. However, because performance was top priority when designing this scheme, the encryption is implemented in an ad hoc manner and can be easily bypassed with information leakage. To protect the exit handler, we use HDFI to enforce that it is always registered with `sdset1` and loaded with `ldchk1`. Since attackers cannot tag an exit handler with `1`, they cannot abuse it to execute arbitrary code.

5.5.3 VTable Pointer Protection

As virtual function calls comprise a large portion of indirect control transfer in large C++ programs like browsers [197], virtual function table pointers (a.k.a., `vfptr`) have become a popular attack target [232]. In these attacks, attackers try to exploit memory corruption vulnerabilities to control the `vfptr` so as to invoke arbitrary code, which has been demonstrated to be very powerful [167]. For this reason, many systems have been proposed to defeat such attacks [97, 197, 232, 230, 27].

Leveraging HDFI, we also implemented a protection mechanism based on one security invariant: *only a constructor function can initialize a* `vfptr`. This invariant can be enforced in two simple steps: (1) when initializing a C++ object, we use `sdset1` to initialize its `vfptr`; and (2) when performing a virtual call, we always use `ldchk1` to load the `vfptr`.

Compared with existing protection mechanisms, our implementation is much simpler in that it requires no sophisticated static analysis and/or runtime instrumentation. At the same time, it is also very effective. More specifically, there are two typical attacks against VTable: injection attacks and reuse attacks. In VTable injection attacks, attackers try to forge a `vfptr` pointing to a crafted VTable. With our protection, this is no longer feasible because the values assigned to `vfptr` are always static/constant. In VTable reuse attacks, attackers try to make the `vfptr` point to an existing VTable, but usually at a wrong offset [167]. Although our mechanism cannot fully prevent all VTable reuse attacks, it significantly increases the difficulty of attacks, because (1) making the `vfptr` point to a wrong offset is no longer feasible, because constructors always assign the correct value; and more importantly, (2) crafting a counterfeit object is also much more difficult, i.e., once combined with techniques that can prevent illegal jumping to the middle of a function (e.g.,

shadow stack and CPS), the only way to modify the `vfptr` is to invoke a constructor, who will initialize a legitimate object and overwrites the crafted data from attackers.

5.5.4 Code Pointer Separation

Control flow hijacking is one of the most popular and powerful attacks. In all control flow hijacking attacks, attackers seize control by corrupting one or more code pointers. Based on this observation, researchers have proposed code pointer separation (CPS) [108], a technique that isolates code pointers into a safe region to prevent attackers from tampering with them. In their original implementation, the isolation is enforced using segment on 32-bit x86 processors or randomization (or masking) on 64-bit x86 processors and ARM processors. As discussed in §5.1, these approaches introduce (1) additional memory overhead for data shadowing, and (2) additional performance overhead for shadow data lookup, which is very problematic on benchmarks where code pointer dereference is more frequent, such as C++ programs and language interpreters. Moreover, their randomization-based approach is subject to brute-force attacks [76], and their masking-based approach introduces an additional 5% performance overhead [108].

By utilizing HDFI, we can eliminate all these drawbacks. Specifically, using the same static analysis from CPS, we can identify all code pointers that need to be protected. With this information, instead of instrumenting the target program to load/store code pointers from the safe region with an additional runtime library, we instrument the program to (1) always use `sdset1` instructions to store code pointers, and (2) always use `ldchk1` instructions to load code pointers. Because no other instructions can store code pointers, our approach has the same effectiveness as segments and masking based approaches. However, because there is no additional lookup step(s), the performance of our approach is better when there are many indirect calls.

One drawback of our solution is that we need to add one additional step to tag static code pointers that are initialized by the OS kernel or the dynamic loader, e.g., virtual function pointers in the VTables. For PIE code, we can reuse our modification to the relocation procedure to perform this task.

Table 11: Components of HDFI and their complexities in terms of their lines of code.

Components	Language	Lines of Code		
		Modified	Added	Total
Architecture	Scala (Chisel)	395	1,803	2,198
Assembler	C	-	16	16
Linux Kernel	C	8	52	60
Total		403	1,871	2,274

5.5.5 Kernel Protection

In §4, we presented KENALI, a kernel protection mechanism aims to prevent memory corruption based privilege escalation attacks. Unfortunately, due to the lack of efficient isolation mechanism, our original implementation (§4.7) imposes high performance overhead. As a generic data isolation mechanism, HDFI can also be used to replace those expensive isolation mechanisms thus reduce the performance overhead.

Similar to CPS, porting KENALI to utilize HDFI is straightforward. Specifically, we replace: (1) its randomization-based stack protection with the shadow stack described in §5.5.1; (2) the expensive, context switch-based update operations with `sdset1`; (3) all read to sensitive data with `ldchk1`; (4) global object shadowing with tagging (i.e., similar to function pointers in the VTable, we wrote a small early initialization routine to tag sensitive global object); and (5) we eliminate its complicated object shadowing mechanism.

5.5.6 Information Leak

In all of the above applications, we try to prevent attackers from injecting data into the trusted region, but HDFI can also be used to prevent attackers from reading sensitive data from the trusted region. For example, in the Heartbleed attack [47], attackers exploited a buffer overread vulnerability in the OpenSSL library to steal the private key associated with the website's certificate. To prevent such attacks, we can (1) tag the memory used to store the private key as `1`, (2) replace all legitimate read access to the key with `ldchk1`, and (3) implement a simple sanitation routine that uses `ldchk0` to check if the buffer to be written to network contains any data with tag `1`.

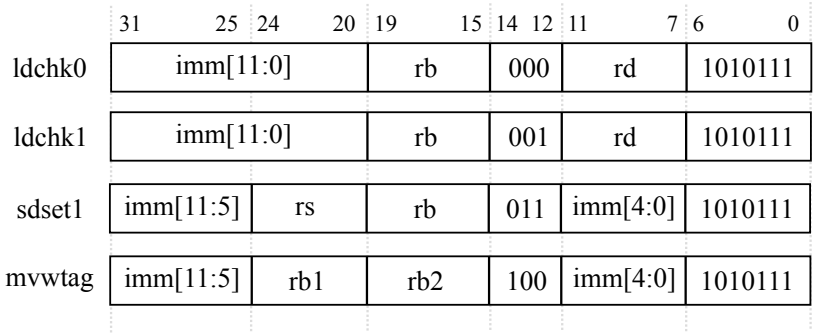

Figure 12: Encoding of HDFI's new instructions.

5.6 Implementation

In this section, we provide the implementation detail of HDFI. Table 11 shows the lines of code used to implement HDFI, excluding empty lines and comments.

5.6.1 Hardware

We implemented a prototype of HDFI by modifying the Rocket Chip Generator [199]. The generated system includes a Rocket Core [200] as its main processor, which has 16KB of L1 instruction and data caches. Modifying the generator itself instead of a generated instance allows us to generate and evaluate multiple versions of HDFI with various features and parameters, e.g., different optimization techniques and configuration parameters.

ISA Extensions.

Following the design pattern of RISC-V, we assign a new opcode to our new instructions that is similar to the RV64I load/store instructions [211]. One difference is `funct3` (bit `[14:12]`) to distinguish four instructions HDFI introduce, unlike RV64I load/store instructions use this field to indicate operand width. Figure 12 shows the new instructions and their encoding format introduced by HDFI.

`sdset1`: We extend the memory request unit's data field by one-bit to include the tag. To determine whether the tag should be `0` or `1`, we introduce a new configuration to the set of control signals for memory command type that is unique to `sdset1`.

`ldchkx`: We add a new, one-bit field to the memory response unit for the tag bit loaded with the machine word. To determine whether the tag bit should be loaded, we assigned a new memory command to these two instructions. Upon a valid response from cache, HDFI

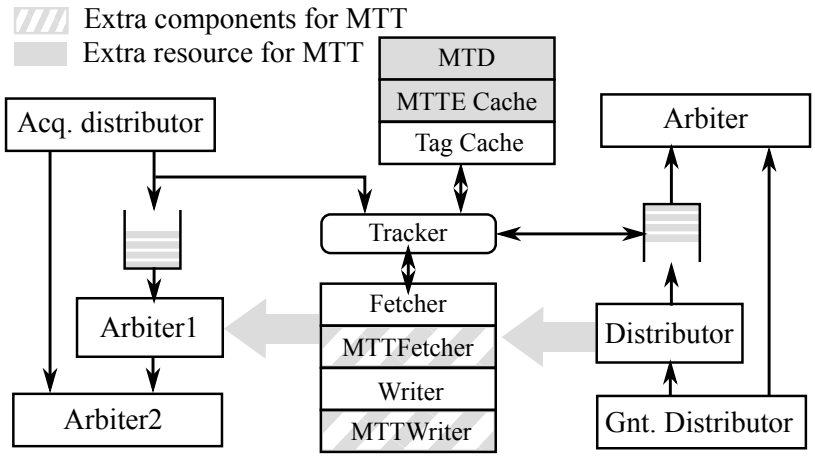

Figure 13: A simplified diagram of DFITAGGER on a Rocket Chip.

compares the tag to the expected value. This expected value is extracted from bit `12` of the `ldchkx` instruction. A tag mismatch generates a new memory exception; otherwise, the pipeline continues normally.

`mvwtag`: At the execution stage, HDFI first calculates the source address from the second register's value and the immediate offset using the ALU, and sends out a memory read request to load the data and tag. The result is stored in a new internal register that is capable of storing both data and tag. Simultaneously, HDFI calculates the destination address from the destination register's value and the same offset using a separate adder. Finally, we issue a memory store request to store the internal register's data and tag to the destination address.

DFITagger. To avoid adding the tag bits physically to the main memory, which is usually a set of DRAMs, we implemented DFITAGGER to translate memory accesses with tags from inner caches into data accesses and tag accesses. Figure 13 shows the DFITAGGER we implemented for the Rocket Chip. The DFITAGGER is designed to handle the memory accesses that comply with the *TileLink* protocol which the rocket chip uses to implement the cache coherence interconnect. Among the five channels that the protocol defines, DFITAGGER handles two of them because they are used to connect the L2 caches and the outer memory system.

To initiate a memory access, the inner cache generates one or more *beats* of transaction through the *Acquire* channel, and the DFITAGGER selectively intercepts the beats using the

Acquire Distributor. When the option `tagger` is enabled, the Acquire Distributor bypasses the device accesses, drops the access to the tag table or meta tag table (for protection purpose), and forwards all the transactions heading to the memory to the *Acquire Queue*, which simply forwards the incoming transactions to memory. The *Acquire Arbiter1* drops all the tag bits in the transactions; the resulting memory accesses only contain the data part of the incoming accesses.

In the meantime, the *Tracker* duplicates the required field of incoming transactions, including the tag bits, the transaction id, the type of the transaction, and the address. When the incoming transaction writes to memory, the Tracker updates the corresponding tag bits in the tag table with the tag bits in the transaction. To do this, the Tracker first check the *Tag Cache*, and uses the *Fetcher* and the *Writer* modules to fetch and evict tag table entries.

Handling memory read accesses is similar, but the Tracker need to intervene in the *Grant* channel as well. In the Rocket Chip, the memory interface (which is a protocol converter) uses the Grant channel to provide the caches with the read data. To attach the tag bits to the Grant transactions, the Acquire Queue changes the transaction id of read accesses so that the corresponding Grant transactions are forwarded to the *Grant Queue*. In the meantime, the Tracker accesses the Tag Cache and uses other modules to prepare the corresponding tag bits. Once the tag bits become available, the Grant Queue forwards the transaction from the memory interface, after changing the transaction id back to the original one and attaching the correct tag bits for each machine word.

Tag Valid Bits. To reduce the number of tag table accesses, HDFI adds a TVB for each machine word in the caches. Using TVB, the cache can avoid fetching the tag bits when it refills a cache line. To take advantage of this, the cache uses the `union` field of an Acquire transaction to mark if the response to the transaction should have valid tag bits or not. The Acquire Distributor then uses this field to decide whether a transaction could be directly forwarded to the *Acquire Arbiter2* and bypass the Acquire Queue.

The location of TVBs is also important. A simple solution is to put the TVBs in the metadata array, where the cache holds the *cache tags* and the coherence information. However, this approach would increase the latency of write hits because the cache has to

update the metadata for every write operation. To address this issue, we choose to put a *tag fetched bit* in the metadata array for each cache line and extend the size of the data array to store the TVBs for each word. The tag fetched bit is set/cleared by the miss handler, which is called *MSHR* in the Rocket Chip. When the handler fetches the cache line with tags, it sets the bit; otherwise the bit is cleared. Since every write operation should update the tag, the cache also sets the TVB whenever a machine word is written.

Adding TVBs also requires the DfiTagger to consider a memory write access whose tag bits are partially valid. To handle this, the cache attaches the TVBs for each machine word to the Acquire transactions for memory writes. With the TVBs, the DfiTagger can selectively update the tag bits in the corresponding tag table entry.

An important drawback of this implementation is that the cache refills a cache line to handle an incoming load with tag access even when the TVB of the requested machine word is set, but if the Tag Fetched Bit is not set. We believe that we can avoid these cache refills by augmenting the miss handler, by letting it to consider the TVBs before evicting and refilling the cache, but the current implementation does not include such feature.

Meta Tag Table. Enabling the Meta Tag Table adds the shaded components and resource in Figure 13 to the DfiTagger. When handling an incoming tag table read access, the Tracker checks whether the MTT cache and the tag cache has a matching entry. If the Tracker fails to find a matching tag table entry, it checks the MTD and the matching MTT entry (loaded into MTT cache if does not exist) to see if the corresponding tag table entry is all zero. If so, the Tracker handle the incoming tag table access without really fetching the entry from the memory. To minimize the miss penalty, the *MTTFetcher* and the *MTTWriter* handles the access to the MTT in the memory in parallel with the existing Writer and Fetcher.

After updating the tag table entry and the MTT entry, the Tracker checks if it can clear the corresponding MTT entry bit and MTD bit. In particular, the Tracker clears the corresponding bit in MTT entry if the updated tag table entry is filled with zeros, and clears the MTD bit if the MTT entry is filled with zeros.

Table 12: Required efforts in implementing or porting security schemes in terms of lines of code.

Solutions	Language	LoC
Shadow Stack	C++ (LLVM 3.3)	4
VTable Protection	C++ (LLVM 3.3)	40
CPS	C++ (LLVM 3.3)	41
Kernel Protection	C (Linux 3.14.41)	70
Library Protection	C (glibc 2.22)	10
Heartbleed Prevention	C (OpenSSL 1.0.1a)	2

5.6.2 Software Support

To utilize HDFI, we made the following changes to the software.

Assembler. We modified the GNU assembler (`gas`) so that it recognizes the new instruction extension and can generate the correct binary.

Kernel Support. Our modifications to the OS kernel include three parts. First, we modified its exception handler to recognize the new tag mismatch exception. To handle this exception, we reused the same logic as normal load/store faults, i.e., generate a segment fault (`SIGSEGV`) for user mode applications, and panic if the exception happens in kernel space. Second, as mentioned in §5.4, we implemented a special memory copy routine with the new `mvwtag` instruction and modified the CoW handler to invoke this routine to copy page content, so that the tag information are preserved. Last, we added routines to allocate the tag table and meta tag table, and initialize the DFITAGGER with the base addresses of the tables.

5.6.3 Security Applications

Most security applications mentioned in §5.5 were implemented based on the llvm-riscv toolchain [166] (RISCV branch). Table 12 summarizes the effort of implementation/porting.

LLVM Shadow Stack. LLVM-based shadow stack is implemented as part of the frame lowering process. Specifically, we modified the `getLoadStoreOpcodes` function to return the opcode of `sdset1` for the `storeRegToStackSlot` function; and return the opcode of `ldchk1` for the `loadRegFromStackSlot` function.

VTable Pointer Protection. VTable pointer protection is implemented in two steps. First, during compilation, we enable the TBAA (type-based alias analysis) option so

Clang will annotate VTable load/store operations with corresponding TBAA metadata (``vtable pointer''). This metadata will be propagated to machine instruction, so in the second step, we leveraged the DAG to DAG transformation pass to replace `sd` instructions with `sdset1` instructions, and to replace `ld` instructions with `ldchk1` instructions, if the machine instruction has the corresponding TBAA of VTables.

Code Pointer Separation. To port CPS [109] to our architecture, we performed the following modifications. (1) Because code pointers are now protected by HDFI, we removed the runtime library required by its original implementation. (2) We modified the instrumentation, so when a code pointer is stored to or loaded from memory, we annotate the corresponding operations with a special TBAA metadata and removes the original invocation to the runtime library. (3) Using the same DAG to DAG transforming function, we replace the `sd` and `ld` instructions with `sdset1` and `ldchk1`, respectively. Unfortunately, lacking link time optimization support in the llvm-riscv toolchain, we cannot port the original CPS and CPI implementations.

Kernel Protection. Due to the limitation of llvm-riscv toolchain, even though we were able to generate LLVM bitcode for the target kernel and apply the static analysis of KENALI, we cannot use Clang to compile the kernel into executable binary. As a result, we cannot perform automated instrumentation to protect all the discover data. For proof-of-concept, we utilize the analysis results to manually instrumented the kernel to protect the `uid` fields in the `cred` structure, which are the most popular target for kernel exploits. Since we have implemented the shadow stack in GCC, we were able to replace KENALI's randomization-based stack protection with our stack shadow.

The rest of the protection mechanisms are implemented through manual modification.

Standard libraries. To protect the integrity of saved context of `setjmp`/`longjmp`, we modified `setjmp.S` and `__longjmp.S` so general registers are saved with `sdset1`, and restored with `ldchk1` to enforce its integrity. To protect the integrity of heap metadata, we manually modified the linking and unlinking routine to use `sdset1` for assigning pointers and `ldchk1`

for loading pointers. To set the tag of static code pointers to `1`, we modified the dynamic loader (`elf_machine_rela`) so that during the relation process, it stores the patched code pointer with tag `1`. And to protect code pointers in GOT table and the exit handler, we modified the dynamic loader to use `sdset1` to set these pointers, and `ldchk1` to load these pointers.

Heartbleed. To protect sensitive data from Heartbleed attacks, we modified OpenSSL so that (1) the private key is stored with `sdset1`; and (2) when building the response buffer, `ldchk0` is used to make sure that all content copied to this buffer has tag `0`. To implement this protection, we used background knowledge about Heartbleed to decide where to put the checking routine (i.e., when constructing the response buffer). For a prototype implementation, we believe this is a reasonable limitation. To thoroughly protect the sensitive data, one could use data flow analysis or taint analysis [221] to determine where to tag sensitive data, and where to put the check.

5.6.4 Synthesized Attacks

To evaluate the effectiveness of the security applications we implemented/ported, we developed/ported several synthesized attacks against different targets.

RIPE Benchmark. RIPE [215] is an open sourced intrusion prevention benchmark. It provides five testbed dimensions: location of the buffer overflow, target code pointers, overflow technique, attack payload and abused function. Since RIPE was developed for the x86 platform, we need to modify it to make it work on the RISC-V architecture. However, due to time limitations, we could not port all the features of RIPE. Specifically, our ported RIPE benchmarks support all locations of buffer overflow, all target code pointers except the frame pointer, both overflow techniques (direct and indirect), one attack payload (return-to-libc), and one abused function (`memcpy`).

Heap Exploit. To evaluate heap metadata protection, we ported the example exploit from [101] to overwrite the return address of a function.

VTable Hijacking. Due to the limitations of the FPGA, we could not use real-world cases like browser attacks to evaluate our VTable pointer protection mechanism. Instead,

we developed a simple attack that overwrites the VTable pointer with a fake one, so the next invocation of the virtual function will invoke the attacker controlled function.

Format String Exploit.

Because the RIPE benchmark does not cover attack targets used in recent attacks, we implemented a simple program with format string vulnerability to evaluate the ported CPS mechanism. We chose a format string vulnerability because it is one of the most powerful vulnerabilities that can be used as local stack read (`%x`), arbitrary memory read (`%s`), and arbitrary memory write (`%n`). For attack targets, we implemented two new attacks: GOT overwriting and atexit handler overwriting.

Kernel Exploit. In the kernel, overwriting non-control data is sufficient to obtain root permissions without hijacking control flow. To test the feasibility of using HDFI to defend against data-only attacks in the kernel, we back ported `CVE-2013-6282` [196], an arbitrary memory read and write vulnerability to our target kernel. Leveraging this vulnerability, an attackers can modify the `uid` of a process and escalate their privilege.

Heartbleed. Heartbleed (`CVE-2014-0160`) [47] is a heap out-of-bounds read vulnerability in OpenSSL caused by missing input validation when parsing malicious TLS heartbeat request. This bug was marked as extremely critical, because researchers have proved that it can be exploited to reveal private keys [86]. To reliably[2] simulate such attacks, we modified vulnerable OpenSSL (`1.0.1a`) to insert special characters as a decoy private key. Since the decoy data is inserted in the affected range of Heartbleed, it can always be leaked in default settings through a Heartbleed attack.

5.7 Evaluation

In this section, we evaluate our prototype of HDFI by answering the following questions:

- **Correctness**. Does our prototype comply with the RISC-V standard (i.e., no backward compatibility issue)? (§5.7.2)

[2]Attacking a OpenSSL-powered HTTPS server cannot always reveal the private key because the buffer used to store the privately may at a lower address, so it cannot be read by a buffer over read.

- **Efficiency**. How much performance overhead does HDFI introduce compared to the unmodified hardware? (§5.7.3)
- **Effectiveness**. Can HDFI-powered security mechanisms accurately prevent attacks? (§5.7.4)
- **Benefits**. Compared to their original implementation, does HDFI-powered implementation perform better and/or is it more secure? (§5.7.5)

5.7.1 Experimental setup

All evaluations were done on the Xilinx Zynq ZC706 evaluation board [217]. The OS kernel is Linux 3.14.41 with support for the RISC-V architecture [149]. Unless otherwise stated, all programs (including the kernel) were compiled with GCC 5.2.0 (`-O2`) and binutils 2.25, with a set of patches to support RISC-V (commit `572033b`) and default kernel configuration of RISC-V. While the board is equipped with 1GB of memory, the Rocket Chip can only use 512MB because the co-equipped ARM system requires 256MB. At boot time, the kernel reserves 8MB for tag tables and 128KB for the meta tag table, respectively. Following the environment that the RISC-V community built, we use the *Frontend Server* that runs on the ARM system and the *Berkeley Boot Loader* that runs on the Rocket Chip to boot vmlinux. The Rocket Chip accesses an ext2 file system in an SD card via the Front-end Server.

Although the tape-out Rocket Core chip can operate on 1GHz or higher, the synthesized FPGA on the ZC706 board can only operate at the maximum frequency of 50MHz. In addition, because the L2 cache is not mature enough for memory-mapped IO [122], we only evaluated with the L1 caches. In place of the L2 cache, we used the *L2BroadcastHub* that interconnects the L1 caches and the outer memory system. Due to the above limitation and the memory limitation of the evaluation board, we were not able to run most SPEC CINT 2006 benchmarks, so we used the much lighter SPEC CINT 2000 [187]. For SPEC CINT 2000, some benchmarks (`gzip` and `bzip`) cannot run successfully with the reference inputs. For these benchmarks, we adjusted the parameters of the reference inputs to reduce the size of the buffer they use to 3MB. We have annotated the results to clarify this.

We used pseudo-LRU (Least Recently Used) as the replacement policy for both tag and

Table 13: Impact of HDFI on memory read latency (ns), with different optimization techniques.

Benchmark	Baseline	Tagger	TVB	MTT	TVB+MTT
L1 hit	40	40 (0%)	40 (0%)	40 (0%)	40 (0%)
L1 miss	760	870 (14.47%)	800 (5.26%)	870 (14.47%)	800 (5.26%)

meta tag caches, and set the size of each cache to 1KB, allowing up to 16 entries of 512-bit cachelines.

5.7.2 Verification

HDFI passes the RISC-V verification suite provided by the RISC-V teams, which means our modifications to the RISC-V complies with the RISC-V standard so unmodified programs can still run correctly on our modified hardware.

5.7.3 Performance Overhead

In this subsection, we evaluate the performance impact of our hardware extension, as well as the effectiveness of our optimization techniques. This evaluation includes two part: the impact of new instructions on the processor core and the impact on memory access. Since HDFI did not introduce many changes to the pipeline of the processor core, the focus will be on memory access.

Pipeline. The `sdset1` and two `ldchk` instructions are treated identically to their normal store and load counterparts in the pipeline, with the exception of `ldchk` doing a comparison at the end of the memory stage. These three instructions can stall the pipeline in the same manner as their counterparts. However, the special register dedicated to `mvwtag` for preserving tags introduces a structural hazard to the pipeline. Because there is only one special register available, a series of `mvwtag` instructions have to wait for the previous `mvwtag` to finish, stalling the pipeline. Other memory instructions do not have to wait on previous ones to issue memory requests.

Memory Access. While the ISA extension does not affect the performance of the processor core, HDFI inevitably introduces additional memory accesses to fetch/update the tag table.

Micro benchmark. To measure the performance impact of these additional memory

Table 14: Impact of HDFI on memory bandwidth (MB/s), with different optimization techniques.

Name	Baseline	Tagger	TVB	MTT	TVB+MTT
Copy	1081	939 (13.14%)	1033 (4.44%)	953 (11.84%)	1035 (4.26%)
Scale	857	766 (10.62%)	816 (4.79%)	776 (9.45%)	817 (4.67%)
Add	1671	1598 (4.37%)	1650 (1.26%)	1602 (4.13%)	1651 (1.2%)
Triad	818	739 (9.66%)	802 (1.96%)	764 (8.8%)	803 (1.83%)

Table 15: Performance overhead of a subset of SPEC CINT 2000 benchmarks.

Benchmark	Baseline	Tagger	TVB	MTT	TVB+MTT
164.gzip	963s	1118s (16.09%)	984s (2.18%)	1029s (6.85%)	981s (1.87%)
175.vpr	14404s	18649s (29.51%)	14869s (3.26%)	15513s (7.71%)	14610s (1.43%)
181.mcf	8397s	11495s (36.89%)	8656s (3.08%)	9544s (13.66%)	8388s (−0.11%)
197.parser	21537s	25005s (16.11%)	22025s (2.27%)	23177s (7.61%)	21866s (1.53%)
254.gap	4224s	4739s (12.19%)	4268s (1.04%)	4500s (6.53%)	4254s (0.71%)
256.bzip2	716s	820s (14.52%)	735s (2.65%)	742s (3.63%)	722s (0.84%)
300.twolf	22240s	28177s (26.71%)	22896s (2.97%)	23883s (7.37%)	22323s (0.36%)

accesses and the logics to deal with them, we used `lat_mem_rd` from LMBench [127] to measure memory access latency and STREAMBench [125] to measure memory bandwidth. Table 13 shows the result of the five configurations. The first row shows that HDFI does not affect the cache access latency. As the system operates at 50MHz, the 40ns latency means that it takes two clock cycles to read from the L1 cache. The second column shows that HDFI does increase the memory access latency. When `TVB` is enabled, DFITAGGER simply bypasses the incoming memory read access unless it explicitly requests the tag bits. However, the access should be examined by the Acquire Distributor and the Grant Distributor (Figure 13), which adds 2 clock cycles latency. For memory bandwidth, our results also show that the optimizations we implemented can effectively reduce overhead.

SPEC CINT 2000. In addition to the micro benchmarks, we also ran a subset of SPEC CINT 2000 benchmarks on the five configurations of HDFI, *without* any security applications (i.e., no load check and no `sdset1`). Due to the limited computing power of the Rocket Chip on FPGA, we chose relatively lighter benchmark. In addition, to be fair, we included relatively memory bound benchmarks. According to a paper [96], 181.mcf, 175.vpr and 300.twolf are memory bound and showing higher overhead. We used reduced version of reference input to run 164.gzip and 256.bzip2.

Table 15 shows that even though the unoptimized version of HDFI causes non-negligible

Table 16: The number of total memory read/write access (MB) from both the processor and DFITAGGER.

Benchmark	T	Baseline	Tagger	TVB	MTT	TVB+MTT
164.gzip	**R**	590	799 (35.25%)	606 (2.71%)	589 (−0.17%)	588 (−0.34%)
	W	380	1,217 (220.26%)	453 (19.21%)	1,017 (167.63%)	378 (−0.53%)
175.vpr	**R**	9,816	17,200 (75.15%)	10,930 (11.35%)	9,760 (−0.57%)	9,792 (−0.25%)
	W	7,908	37,480 (373.83%)	12,420 (57.06%)	31,890 (303.16%)	7905 (0%)
181.mcf	**R**	9,778	14,310 (46.35%)	10,503 (7.41%)	9,778 (0%)	9,778 (0%)
	W	5,588	23,720 (324.33%)	8,490 (1.11%)	20,300 (263.15%)	5,588 (0%)
197.parser	**R**	12,770	17,610 (37.9%)	13,220 (3.52%)	12,850 (0.63%)	12777 (0.01%)
	W	8,290	27,490 (231.6%)	9,640 (16.28%)	24,440 (194.81%)	8299 (0.11%)
254.gap	**R**	2,233	2,872 (28.61%)	2,239 (0.27%)	2,225 (0%)	2,206 (−1.21%)
	W	1,594	4,237 (165.81%)	1,701 (6.71%)	3,926 (146.3%)	1,592 (−0.13%)
256.bzip2	**R**	228	390 (71.05%)	268 (17.54%)	229 (0.44%)	229 (0.44%)
	W	249	896 (259.84%)	407 (63.45%)	730 (193.17%)	249 (0%)
300.twolf	**R**	13,600	22,350 (64.34%)	15,820 (16.32%)	13,600 (0%)	13,610 (0%)
	W	13,680	48,650 (255.63%)	22,510 (64.55%)	38,090 (178.43%)	13,610 (−0.51%)

performance overhead, our optimizations successfully eliminated a large portion of overhead. Specifically, since there is no load check, TVB eliminated all read access requests to the tag table; and since there is no `sdset1`, MTT eliminated all the write access to the tag table. Table 16 shows the number of memory accesses reduced by TVB and MTT. Please note that the 0.11% performance gain on `mcf` is due to fluctuations.

5.7.4 Security Experiments

In this subsection, we evaluate the effectiveness of HDFI-powered protection mechanisms. We evaluated all the security applications described in §5.5, with synthesized attacks described in §5.6.4. The evaluation result is shown in Table 17, all HDFI-powered protection mechanisms can successfully mitigate the corresponding attack(s).

RIPE benchmark. With our ported RIPE benchmark, there are 112 possible combinations, with 54 that could proceed and 58 are not possible. Please note that although we did not port all combinations, all attack targets are supported except the frame pointer, which behaves quite differently on RISC-V. The supported targets are: return address, stack function pointer, heap function pointer, `.bss` section function pointer, `.data` section function pointer, `jmp_buf` on stack, `jmp_buf` as stack parameter, `jmp_buf` in heap, `jmp_buf` in `.bss` section, `jmp_buf` in `.data` section, function pointer in a structure on stack, in heap, in `.bss` section and in `.data` section. With our ported CPS, we can prevent all 54 attacks.

Table 17: Security evaluation of applications utilizing HDFI.

Mechanism	Attacks	Result
Shadow stack	RIPE	✓
Heap metadata protection	Heap exploit	✓
VTable protection	VTable hijacking	✓
Code pointer separation (CPS)	RIPE	✓
Code pointer separation (CPS)	Format string exploit	✓
Kernel protection	Privilege escalation	✓
Private key leak prevention	Heartbleed	✓

Heap exploit. Without protection, our basic version of heap attack targeting newlibc (a lightweight libc) was able to overwrite the return address to launch a return-to-libc attack to invoke the "evil" function. With our enhanced library, we were able to stop the attack.

VTable hijacking. Without protection, our simple VTable hijacking attack was able to invoke the "evil" function. With our VTable protection mechanism, we were able to prevent the loading of attacker-crafted `vfptr`.

Format string exploit. Without protection, our format string exploit can overwrite the GOT table entry and the exit handler to invoke the "evil" function. With our enhanced library, both attacks were stopped.

Kernel exploit. Without protection, the exploit can change the `uid` of the attack process to a arbitrary number. With our protection, the attack causes a kernel panic when trying to access the `uid`.

Heartbleed. : without protection, we can leak the decoy secret by exploiting the Heartbleed vulnerability. With our protection, the attack was stopped when constructing the response buffer.

5.7.5 Impact on Existing Security Solutions

As a fine-grained hardware-based isolation mechanism, we expect HDFI to provide the following benefits:

I **Security**: HDFI should provide non-bypassable protection for the isolated data;

II **Efficiency**: HDFI should provide the protection with low performance overhead;

III **Elegance**: HDFI should enable the building of elegant security solutions, e.g., no data shadowing, which as discussed in the introduction, has many drawbacks;

IV **Usability**: HDFI should be flexible, capable of supporting different security solutions; it should also be easy to use, so as to increase the chance of real-world adoption.

In this subsection, we evaluate whether HDFI achieves these design goals or not. As described in §5.5, none of the HDFI-powered security applications requires data shadowing, including three solutions (stack protection, CPS and KENALI) whose previous implementations rely heavily on data shadowing. For this reason, we consider HDFI to have achieved goal III. And as shown in Table 12, implementing/porting security solutions with HDFI is very easy, so we consider goal IV to be achieved as well. Next, we analyze the security and efficiency benefit.

Security Improvement. Compare with software-based shadow stacks [60], our stack protection provides better security than platforms that do not have efficient isolation mechanisms, such as x86_64 and ARM64. Compared with existing hardware-based shadow stacks [218, 114, 150], our solution provides the same security guarantee but is more flexible and supports kernel stack. Compared to active callsite based solutions [64, 63], our stack protection provide better security. For standard libraries, existing heap metadata integrity checks can be bypassed under certain conditions. For example, Google project zero team has successfully compromised `ptmalloc` with NULL off-by-one [75]; and existing encryption-based exit handler protection is vulnerable to information leak based attacks. However, Our HDFI-based library enhancement cannot be bypassed because attackers cannot control the hardware-managed tags. Compared with existing VTable protection mechanisms [97, 197, 232, 230, 27], our HDFI-based solution has both advantages and limitations. On the positive side, our approach makes it much harder to overwrite the `vfptr`; while in all other solutions, attackers can easily tamper with `vfptr`. However, because our approach does not involve any class hierarchy analysis, we cannot guarantee type safety (i.e., semantic correctness). Compared to the original CPS implementation, our ported version provides the same security guarantee as segment-based isolation but is stronger than its

Table 18: LMBench results of baseline system and HDFI with kernel stack protection.

Benchmark	Baseline	Kernel Stack Protection
null syscall	8.91μs	8.934μs (0.27%)
open/close	160.6μs	168.7μs (5.04%)
select	285.6μs	287.5μs (0.67%)
signal install	19.3μs	21.5μs (11.4%)
signal catch	99.8μs	105.6μs (5.81%)
pipe	273.6μs	306.6μs (12.06%)
fork+exit	5892μs	6308μs (7.06%)
fork+execv	6510μs	6972μs (7.1%)
page fault	50.0μs	52.6μs (5.2%)
mmap	800μs	880μs (10%)

randomization-based isolation, which has been proven to be vulnerable [76]. Compared to the original implementation of KENALI §4.7, our ported version provides stronger guarantees than its randomization-based stack. Based on the above analysis, we also consider HDFI to achieve goal I.

Performance Improvement. Because we can neither fully port the original implementation of CPS and KENALI to our testbed due to problems with the official llvm-riscv toolchain nor run the C++ benchmarks of SPEC CINT 2000, we designed the following benchmarks to evaluate the performance improvement of HDFI-based security solutions.

Micro benchmarks. Compared with the original implementation of CPS, our ported version would be more efficient because it does not need to access the shadow data. To demonstrate this benefit, we implemented a micro benchmark that measures the overhead for performing an indirect call for 1,000 times. To simulate CPS, we used their own hash table implementation and performed the same look up before the indirect call. For our implementation, we just replaced the load instruction with a checked load. Note, although our implementation sounds simpler, it provides the same level of security guarantee as the original segment-based CPS implementation. The result showed that our protection only incurs 1.6% overhead, whereas the hash table lookup incurred 971.8% overhead. Note, this micro benchmark only shows the worst case performance of both approaches. Depending on the running application, the real end-user performance impacts could be much less than this.

Because we cannot perform automated instrumentation to fully replicate KENALI, here we only measured the performance overhead of kernel stack protection. The result is shown

Table 19: Performance overhead of HDFI-based shadow stack CPS.

Benchmark	GCC	Shadow Stack		Clang	CPS+SS	
164.gzip	981s	992s	(1.12%)	1734s	1776s	(2.42%)
181.mcf	8388s	8536s	(1.76%)	11014s	11403s	(3.54%)
254.gap	4254s	4396s	(3.34%)	20783s	23526s	(13.23%)
256.bzip2	722s	744s	(3.05%)	1454s	1521s	(4.61%)

in Table 18. Although our prototype implementation has higher a performance overhead, it is also more secure than the randomization-based stack protection used in the original implementation.

SPEC CINT 2000. To measure the performance overhead of HDFI under the existence of load check and store set, we ran four benchmarks from SPEC CINT 2000 with two security protections: GCC-based shadow stack and CPS plus LLVM-based shadow stack. The result is shown in Table 19. As we can see, the performance overhead is also low. Please note that because Clang cannot compile the benchmarks with `-O2`, they are compiled with `-O0`. As a result, the performance is much worse than GCC. More importantly, because Clang did not optimize redundant stack access with `-O0`, it caused trouble for our current implementation of TVB (§5.6.1); this is the reason why the `gap` benchmark behaved so badly on CPS.

5.8 Security Analysis

Being an isolation mechanism, HDFI cannot guarantee memory safety by itself, so it cannot prevent all memory corruption-based attacks. In this section, we analyze the security guarantee provided by HDFI and provide our recommendations on how to utilize HDFI properly in security solutions.

5.8.1 Attack Surface

The security guarantee of HDFI is in data-flow isolation, i.e., preventing data flowing from one region to another. This is enforced by (1) partitioning write operations into two groups: those who can set the memory tag to `1`, and those who set the memory tag to `0`; and (2) when loading, checking if the tag matches the expected value. In this regard, HDFI has the following attack surfaces:

Inaccuracy of Data-flow Analysis. The first challenge for utilizing HDFI is how to

correctly perform partitioning and checking. To do so, we rely on data-flow analysis. For some security-critical data, such as return addresses and VTable pointers, their data-flow is quite simple, so the accuracy can be easily guaranteed even without any program analysis. For data like code pointers, because their data-flow is more complicated, it would require thorough static analysis to guarantee the accuracy. Fortunately, because these data are usually self-contained, i.e., not provided by external input, the accuracy, to some extent, can still be guaranteed. However, for data that exhibits complicated data-flow, it may not always be possible to guarantee the accuracy of static analysis. In this case, the common strategy is to avoid false positives by allowing false negatives, i.e., allowing some attacker controllable write operations to set the memory tags. As a result, HDFI itself is not sufficient to guarantee data integrity, so one must employ other runtime protection techniques to compensate for such inaccuracies.

Deputy Attacks. After partitioning, the next challenge is how to guarantee the trustworthiness of each write operation. More specifically, a write operation takes two parameters, a *value* and an *address*. The integrity of a write operation thus relies on the integrity of both the value and the address. If either of them can be controlled by attackers or the instruction gets executed under wrong context (e.g., via control flow hijacking), then they can launch *deputy attacks*. Please note that the control here means both *direct* and *indirect* control. For example, if attackers can control the object pointer used to invoke a C++ constructor, then even though our VTable pointer protection can prevent them from directly overwriting the VTable pointer, they can still leverage this constructor to overwrite the VTable pointer of an existing C++ object. Similarly, if a piece of sensitive data may propagate from one memory location to another, and one forgets to check the tag of the source before setting the tag of the destination to `1`, then an attack can leverage this bug to overwrite sensitive data with a value controlled by the attacker.

5.8.2 Best Practices

To mitigate the aforementioned attacks, we recommend utilizing HDFI in the following ways:

(1) To prevent write operations from executing under the wrong context, it is important

to enforce the integrity of the control flow, which is also required by other systems that enforces write capability [34, 4]. With HDFI, this can be easily achieved through protecting all the control data (e.g., CPS + shadow stack).

(2) To prevent attackers from controlling the address parameter of write operations, it is important to recursively protect all pointers that are part of the dereference chain [108, 182]. It is worth noting that because HDFI is designed to be fine-grained and its protection is enforced efficiently by hardware, including more pointers would not be a big performance issue.

(3) To prevent attackers from controlling the value parameter of write operations, one must ensure that the value is trusted. A value is trusted if any of these conditions hold: (1) it is a constant; (2) it is from a trusted register (e.g., the link register); (3) it is loaded from a memory location with the expected tag; or (4) the semantic of the current program context guarantees the trustworthiness of the value (e.g., during early kernel initialization or when the program is being initialized by the dynamic loader). Moreover, if the value may have both tags (e.g., unions in C), one should use the special memory copy instruction to propagate data with the tag when the data is not modified or leverage a exception handler when the data needs to be modified between load and store.

(4) To compensate the potential inaccuracy of data-flow analysis, we recommend combining HDFI with a runtime memory safety enforcement mechanism like [134, 94]. By doing so, even if we allow attackers to control some write operations, the memory safety protection mechanism would prevent attackers from abusing those write operations to launch attacks.

5.9 Limitations and Future Work

5.9.1 DMA Attacks

Since our current prototype of HDFI only handles memory accesses from the processor core, it is vulnerable to DMA-based attacks. Attackers can leverage DMA to (1) corrupt the data without changing the tag and (2) directly attack the tag table. To mitigate this threat, we could leverage features like IOMMU to confine the memory that can be accessed through DMA [173]. Alternatively, we can choose to add our own hardware module in between the

interconnect and the memory controller such that all memory accesses would pass through the hardware module. By doing so, our hardware module would be able to determine whether or not the access is from DfiTagger, thus prevents malicious access to the tag table. It is worth noting that similar hardware modules have already been introduced [132] and deployed in commodity hardware [94, 11].

5.9.2 Configurable Tag Table

Our current implementation completely blocks accesses to the tag table. Although this provides a stronger security guarantee, it also comes with some drawbacks. The first problem is that we cannot save the page to disk because the tag information will lost. To support these features, we must allow the kernel to access the tag table. However, to protect the tag table from tampering, we must implement some protection techniques like [62] or integrity measurements like [94]. Another drawback of our current design is that we must allocate the whole tag table in advance. In the future, we could provide other options for the OS kernel or the hypervisor to manage the tag table depending on the security requirement by users. On such a model, we can implement an on-demand allocation mechanism to reduce the memory overheads, i.e., we allocate the tag memory only when DfiTagger modifies a tag entry.

5.9.3 Further Optimizations

Although the Rocket Chip Generator is a great tool for prototype verification, the Rocket Core is a very limited processor compared to x86 processors. With a more powerful processor core like the Berkeley out-of-order machine (BOOM) [36] and a more sophisticated cache, we could further reduce the memory access overhead using the following techniques.

Tag Prefetch. Just like prefetching data that is likely to be used in the future due to program locality, we could also prefetch the tag. We could both prefetch the tag from DfiTagger to avoid possible read miss hit due to TVB and prefetch the tag entries from the main memory when the bus is free.

Delayed Check. Just like speculating a branch, as most tag checks should not triggering

the exception, with an out-of-order machine we could speculate the execution even when the tag is not ready (i.e., TVB miss hit). By doing so, we could avoid stalling the pipeline and further reduce the overhead of HDFI.

Better Cache Design. In our prototype implementation, we did not extend our modification to the L2 cache. At the same time, as mentioned in §5.6.1, out current design of TVB is not ideal, which may cause some obvious performance overhead for unoptimized programs (§5.7.5). For future work, we plan to extend our modification to the L2 cache with better TVB implementation.

5.9.4 Dynamic Code Generation

Dynamic code generation is an important technique that has been widely utilized in browsers and OS kernels to improve performance. However, because this technique requires memory to be both writable and executable, it may be vulnerable to code cache injection attacks (§3); and unlike static code, it is not always possible to detect malicious modification to the generated code. In the future, we can perform tag checking for instruction fetching, i.e., provide a configuration flag that once enabled, only allows tagged memory to be fetched as code.

5.10 Summary

In this chapter, we have presented HDFI, a new fine-grained data isolation mechanism. HDFI uses new machine instructions and hardware features to enforce isolation at the machine word granularity, by virtually extending each memory unit with an additional tag that is defined by data-flow. To implement HDFI, we extended the RISC-V instruction set architecture and instantiated it on the Xilinx Zynq ZC706 evaluation board. Our evaluation using benchmarks including SPEC CINT 2000 showed that the performance overhead due to our hardware modification is low ($< 2\%$). We also implemented security mechanisms including stack protection, standard library enhancement, virtual function table protection, code pointer protection, kernel data protection, and information leak prevention on HDFI. Our results show that HDFI is easy to use, imposes low performance overhead, and improves security.

CHAPTER VI

CONCLUSION

6.1 Summary

Exploits against memory corruption vulnerabilities is one of the most popular attack vector to compromise computer systems. Despite much effort spent on preventing such attacks, existing solutions still suffers from two main limitations. That is, solutions that can provide strong security guarantees are too slow for practical deployment and efficient solutions can only provide limited security guarantee against this threat. This dissertation proposed several new techniques that advanced the state-of-the-art on defending against memory-corruption-based exploits.

First, we proposed SDCG, a novel system design which enables dynamic code generation to comply with the W⊕X policy and fundamentally eliminated the possibility of code injection attacks. We implemented SDCG for two types of popular code generators: JS engine (V8 [85]) and DBT (Strata [169]). Our implementation experience showed that porting code generators to SDCG only requires a small modification. The security and performance evaluation of our two prototype implementations showed that SDCG is secure under our threat model and the performance overhead is small: around 6.90% (32-bit) and 5.65% (64-bit) for V8 benchmark suite; and around 1.64% for SPEC CINT 2006 (additional to Strata's own overhead).

Second, we proposed KENALI, a new defense system against memory-corruption-based kernel privilege escalation attacks. KENALI utilizes data-flow integrity (DFI) to enforce two basic security invariants of access control mechanisms—complete mediation and tamper proof. To reduce the performance overhead of DFI enforcement, KENALI leverages two new techniques. The first technique INFERDISTS soundly and automatically infers memory objects that are vital to enforce the two security invariants. The second technique PROTECTDISTS selectively enforces DFI over the inference results. The combination of these two techniques

significantly reduced the performance overhead without sacrificing the security guarantees. For demonstration, we implemented a prototype of KENALI that protects the Linux kernel for the 64-bit ARM architecture that powers Android devices, for its popularity and long update cycle. The security and performance evaluation of our prototype implementation showed that KENALI is able to prevent a large variety of privilege escalation attacks; at the same time, its performance overhead is also moderate, around 7-15% for standard Android benchmarks.

Third and last, we proposed HDFI, a new fine-grained hardware isolation mechanism. HDFI enforces data isolation at machine word granularity by virtually extending each physical address with an additional tag. Inspired by the idea of DFI, HDFI defines the tag of a memory unit by the last instruction that writes to this memory location; then at memory read, it allows a program to check if the tag matches what is expected. This capability allows developers to enforce different security models. For example, to protect the *integrity* of sensitive data, we can enforce the Biba Integrity Model and to enforce the *confidentiality* of sensitive data, we can use it to enforce the Bell–LaPadula Model. We implemented a prototype of HDFI by extending the RISC-V instruction set architecture (ISA) with support of one-bit tag. In order to demonstrate the benefit of HDFI to security solutions, we developed and ported six representative security mechanisms to leverage HDFI, including stack protection, standard library enhancement (protection for `setjmp/longjmp`, heap metadata, GOT, and the exit handler), virtual function table protection, code pointer separation, kernel data protection, and information leak prevention. Compared to existing solution, HDFI-based solutions have several advantages: (1) our development experience shows that HDFI is easy to use and usually allows us to create simpler solutions; (2) as a hardware-enforced isolation mechanism, HDFI can help improve the security guarantees; (3) by eliminating data shadowing and context switching, HDFI can also help reduce the performance overhead for security mechanisms.

6.2 Thesis Contributions

We recap the thesis contributions:

- **New Threats Highlighting** With real exploits, we highlighted the threats of code cache injection attacks and data-oriented attacks.

- **New Software Design** We presented a new software design to resolve the conflict between W⊕X policy and dynamic code generation and to block all code injection attacks.

- **New Program Analysis Technique** The key challenge for building practical defense mechanisms against data-oriented attacks is how to identify data that are vital to those attacks. To solve this challenge, we have developed an automated program analysis technique that can infer data that is critical to kernel privilege escalation attacks.

- **New Isolation Techniques** We developed several techniques to enforce efficient protection over selective memory content and applied them to the Android kernel.

- **New Hardware Design** We developed a new hardware feature to overcome the limitation of the isolation mechanisms provided by commodity hardware. We further demonstrated the benefits of our new hardware feature via developing and evaluating of several defense techniques against memory-corruption-based exploits.

- **Open Source** We will open source all prototype implementations of the techniques presented in this thesis for better real world adoption.

6.3 Future Work

Among the five exploit techniques discussed in §2.2, code injection attacks can be prevented by W⊕X and SDCG, control-flow hijacking attacks can be prevented by the combination of shadow stack and code-pointer protection, and with KENALI and HDFI, we have made promising progress against data-oriented attacks. However, the two remaining exploit techniques still lack enough research.

Information Leak. Information leak, especially generic, memory corruption-based information leak is still a considerable threat to cyber systems. At the same time, preventing this attack is also more challenging, as read operations are usually much more common than

write operations, which means checking reads is likely to impose much higher performance overhead than checking writes. For this reason, developing practical defense mechanisms against generic information leak attacks is still an open and critical research problem.

Data Overlapping. As exploits against use-after-free vulnerabilities have been very popular for the past few years, many techniques have been proposed and deployed to defeat this threat. However, much less effort has been spent on detecting and preventing uninitialized data use. Although most compilers has provided the `-Wuninitialized` option to detect accessing uninitialized stack variable, its scope is limited to current function. As a result, there still exist many uninitialized data use in programs. Moreover, the consequence of uninitialized data use can also be dangerous, ranging from leaking security-critical data to arbitrary code execution. Therefore, more effort should be spent on detecting and mitigating this attack vector.

6.4 Closing Remarks

This thesis documented our research on building principled and practical defense techniques against memory-corruption-base exploits. We first proposed, implemented, and evaluated a novel system design (SDCG) that can block all code injection attacks with small performance overhead. Then we studied the key challenges to defend against data-oriented attacks and proposed, implemented, and evaluated two novel techniques (INFERDISTS and PROTECTDISTS) that can block all memory-corruption-based kernel privilege escalation attacks. Finally, we proposed, implemented, and evaluated a novel hardware feature (HDFI) that enables us to build simpler, more secure, and more efficient defense mechanisms against memory-corruption-based exploits. While my thesis work has advanced the state-of-the-art on the defense against memory-corruption-based exploits and covers the three most popular exploit techniques, there are open problems for future research.

.

REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J., "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Adobe Product Security Incident Response Team, "Inside Adobe Reader Protected Mode." http://blogs.adobe.com/security/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html, 2010.

[3] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M., "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[5] Akritidis, P., Costa, M., Castro, M., and Hand, S., "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security Symposium (Security)*, 2009.

[6] Andersen, S. and Abella, V., "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies." http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[7] Anderson, J. P., "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), 1972.

[8] Android Open Source Project, "Verified boot." https://source.android.com/devices/tech/security/verifiedboot/index.html.

[9] Ansel, J., Marchenko, P., Erlingsson, Ú., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B., "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[10] Arbaugh, W., Farber, D. J., Smith, J. M., and others, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy (Oakland)*, 1997.

[11] ARM, "CoreLink™ TrustZone Address Space Controller TZC-380." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf, 2010.

[12] ARM Limited, *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM Limited, 2015.

[13] Ashcraft, K. and Engler, D. R., "Using programmer-written compiler extensions to catch security holes," in *IEEE Symposium on Security and Privacy (Oakland)*, 2002.

[14] Austin, T. M., Breach, S. E., and Sohi, G. S., "Efficient detection of all pointer and array access errors," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[15] Aycock, J., "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[16] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W., "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[17] Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., and Pewny, J., "You can run but you can't read: Preventing disclosure exploits in executable code," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[18] Baliga, A., Ganapathy, V., and Iftode, L., "Automatic inference and enforcement of kernel data structure invariants," in *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[19] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A., "Thorough static analysis of device drivers," in *European Conference on Computer Systems (EuroSys)*, pp. 73–85, 2006.

[20] Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D., "Randomized instruction set emulation to disrupt binary code injection attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[21] Bell, D. E. and LaPadula, L. J., "Secure computer systems: Mathematical foundations," tech. rep., DTIC Document, 1973.

[22] Bellard, F., "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference (ATC)*, 2005.

[23] Berger, E. D. and Zorn, B. G., "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[24] Biba, K. J., "Integrity considerations for secure computer systems," tech. rep., DTIC Document, 1977.

[25] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D., "Hacking blind," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[26] Bonwick, J. and Moore, B., "Zfs: The last word in file systems," 2007.

[27] Bounov, D., Kici, R., and Lerner, S., "Protecting c++ dynamic dispatch through vtable interleaving," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[28] Bradbury, W. S. A. and Mullins, R., "Towards general purpose tagged memory." `http://riscv.org/workshop-jun2015/riscv-tagged-mem-workshop-june2015.pdf`, 2015.

[29] Bruening, D. and Zhao, Q., "Practical memory checking with dr. memory," in *International Symposium on Code Generation and Optimization (CGO)*, 2011.

[30] Bulba, K., "Bypassing stackguard and stackshield," *Phrack Magazine*, vol. 10, no. 56, 2000.

[31] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X., "Mapping kernel objects to enable systematic integrity checking," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[32] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R., "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security Symposium (Security)*, 2015.

[33] Carlini, N. and Wagner, D., "Rop is still dangerous: Breaking modern defenses," in *USENIX Security Symposium (Security)*, 2014.

[34] Castro, M., Costa, M., and Harris, T., "Securing software by enforcing data-flow integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R., "Fast byte-granularity software fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[36] Celio, C., Patterson, D. A., and Asanović, K., "The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," Tech. Rep. UCB/EECS-2015-167, UCB, 2015.

[37] Chen, H. and Wagner, D., "MOPS: an infrastructure for examining security properties of software," in *ACM Conference on Computer and Communications Security (CCS)*, 2002.

[38] Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., and Kaashoek, M. F., "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Asia-Pacific Workshop on Systems (APSys)*, 2011.

[39] Chen, J., "Andersen's inclusion-based pointer analysis re-implementation in llvm." `https://github.com/grievejia/andersen/tree/field-sens`.

[40] Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P. B., Mowry, T. C., Ramachandran, V., Ruwase, O., Ryan, M., and Vlachos, E., "Flexible hardware acceleration for instruction-grain program monitoring," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008.

[41] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K., "Non-control-data attacks are realistic threats," in *USENIX Security Symposium (Security)*, 2005.

[42] Chen, X., "ASLR Bypass Apocalypse in Recent Zero-Day Exploits." `http://www.fireeye.com/blog/technical/cyber-exploits/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`, 2013.

[43] Cheng, W., Zhao, Q., Yu, B., and Hiroshige, S., "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *IEEE Symposium on Computers and Communications (ISCC)*, 2006.

[44] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H., "Ropecker: A generic and practical approach for defending against rop attacks," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[45] Chiueh, T.-c. and Hsu, F.-H., "Rad: A compile-time solution to buffer overflow attacks," in *ICDCS*, 2001.

[46] Christoulakis, N., Christou, G., Athanasopoulos, E., and Ioannidis, S., "Hcfi: Hardware-enforced control-flow integrity," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[47] Codenomicon and Mehta, N., "The Heartbleed Bug." `http://heartbleed.com/`, 2014.

[48] Common Weakness Enumeration, "Cwe-416: use after free."

[49] Common Weakness Enumeration, "Cwe-680: Integer overflow to buffer overflow."

[50] Conover, M., "w00w00 on heap overflows," 1999.

[51] Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., Negro, M., Qunaibit, M., and Sadeghi, A. R., "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[52] Corbet, J., "Yet another new approach to seccomp." `http://lwn.net/Articles/475043/`, 2012.

[53] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX Security Symposium (Security)*, 1998.

[54] Cowan, C., Beattie, S., Johansen, J., and Wagle, P., "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *USENIX Security Symposium (Security)*, 2003.

[55] Crandall, J. R. and Chong, F. T., "Minos: Control data attack prevention orthogonal to memory model," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[56] Criswell, J., Dautenhahn, N., and Adve, V., "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[57] Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V., "Secure virtual architecture: A safe execution environment for commodity operating systems," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[58] CVE, "CVE vulnerabilities found in browsers." http://web.nvd.nist.gov/view/vuln/search-results?query=browser&search_type=all&cves=on.

[59] Dalton, M., Kannan, H., and Kozyrakis, C., "Raksha: a flexible information flow architecture for software security," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2007.

[60] Dang, T. H., Maniatis, P., and Wagner, D., "The performance cost of shadow stacks and stack canaries," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[61] Daniel, M., Honoroff, J., and Miller, C., "Engineering heap overflow exploits with javascript," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[62] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., and Adve, V., "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[63] Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-R., Koeberl, P., Sullivan, D., Arias, O., and Jin, Y., "HAFIX: Hardware-assisted flow integrity extension," in *Annual Design Automation Conference*, 2015.

[64] Davi, L., Koeberl, P., and Sadeghi, A. R., "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Annual Design Automation Conference*, 2014.

[65] Davi, L., Liebchen, C., Sadeghi, A.-R., Snow, K. Z., and Monrose, F., "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming.," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[66] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F., "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium (Security)*, 2014.

[67] Deng, D. Y. and Suh, G. E., "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *International Conference on Dependable Systems and Networks (DSN)*, 2012.

[68] Designer, S., "Getting around non-executable stack (and fix)." http://seclists.org/bugtraq/1997/Aug/63, 1997.

[69] Devietti, J., Blundell, C., Martin, M. M. K., and Zdancewic, S., "Hardbound: Architectural support for spatial safety of the C programming language," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[70] Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J. M., Knight Jr, T. F., Pierce, B. C., and DeHon, A., "Pump: a programmable unit for metadata processing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2014.

[71] Dhurjati, D. and Adve, V., "Backwards-compatible array bounds checking for c with very low overhead," in *International Conference on Software Engineering (ICSE)*, 2006.

[72] Drepper, U., "Pointer encryption." `http://udrepper.livejournal.com/13393.html`, 2007.

[73] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. X., "Dynamic spyware analysis.," in *USENIX Annual Technical Conference (ATC)*, 2007.

[74] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C., "Xfi: Software guards for system address spaces," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[75] Evans, C. and Ormandy, T., "The poisoned NUL byte, 2014 edition," 2014.

[76] Evans, I., Fingeret, S., González, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H., "Missing the point(er): On the effectiveness of code pointer integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[77] Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S., "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[78] Ferguson, J. N., "Understanding the heap by breaking it," 2007. Black Hat USA.

[79] Ford, B. and Cox, R., "Vx32: Lightweight user-level sandboxing on the x86," in *USENIX Annual Technical Conference (ATC)*, 2008.

[80] Garfinkel, T., Pfaff, B., and Rosenblum, M., "Ostia: A delegating architecture for secure system call interposition," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[81] Gawlik, R., Kollenda, B., Koppe, P., Garmany, B., and Holz, T., "Enabling client-side crash-resistance to overcome diversification and information hiding," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[82] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G., "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[83] Google, "Seccompsandbox." `https://code.google.com/p/seccompsandbox/wiki/overview`.

[84] Google, "The sandbox design principles in Chrome." `http://dev.chromium.org/developers/design-documents/sandbox`.

[85] Google, "Design of chrome v8." `https://developers.google.com/v8/design`, 2008.

[86] Graham-Cumming, J., "Searching for the prime suspect: How heartbleed leaked private keys." https://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys/, 2014.

[87] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W., "ILR: Where'd My Gadgets Go?," in *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[88] Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z., "Automatic generation of data-oriented exploits," in *USENIX Security Symposium (Security)*, 2015.

[89] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z., "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[90] Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D., Knight, J. C., Nguyen-Tuong, A., and Rowanhill, J., "Secure and practical defense against code-injection attacks using software dynamic translation," in *International Conference on Virtual Execution Environments (VEE)*, 2006.

[91] Hund, R., Willems, C., and Holz, T., "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[92] Intel, "Intel kernel-guard technology." https://01.org/intel-kgt.

[93] Intel, "Introduction to intel memory protection extensions." https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[94] Intel Corporate, "Intel architecture instruction set extensions programming reference." https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference, 2013.

[95] Intel Corporation, "Control-flow enforcement." https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2016.

[96] Jaleel, A., "Memory characterization of workloads using instrumentation-driven simulation." http://www.glue.umd.edu/~ajaleel/workload/, 2008.

[97] Jang, D., Tatlock, Z., and Lerner, S., "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[98] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y., "Cyclone: A safe dialect of c.," in *USENIX Annual Technical Conference (ATC)*, 2002.

[99] Johnson, R. and Wagner, D., "Finding user/kernel pointer bugs with type inference.," in *USENIX Security Symposium (Security)*, 2004.

[100] Jones, R. W. and Kelly, P. H., "Backwards-compatible bounds checking for arrays and pointers in c programs," in *International Workshop on Automatic Debugging*, 1997.

[101] Kaempf, M., "Smashing the heap for fun and profit," *Phrack Magazine*, vol. 11, no. 57, 2001.

[102] Kannan, H., Dalton, M., and Kozyrakis, C., "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[103] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D., "Branch regulation: Low-overhead protection from code reuse attacks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[104] Kemerlis, V. P., Portokalidis, G., and Keromytis, A. D., "kguard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security Symposium (Security)*, 2012.

[105] Kil, C., Jim, J., Bookholt, C., Xu, J., and Ning, P., "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[106] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O., "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[107] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., and others, "sel4: Formal verification of an os kernel," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[108] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D., "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[109] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D., "Code pointer integrity (early technology preview)." `https://dslabpc10.epfl.ch/ssl_read/levee/levee-early-preview-0.2.tgz`, 2014.

[110] Lameter, C., "Slub: The unqueued slab allocator." `http://lwn.net/Articles/223411/`, 2007.

[111] Larochelle, D. and Evans, D., "Statically detecting likely buffer overflow vulnerabilities.," in *USENIX Security Symposium (Security)*, 2001.

[112] Lea, D. and Gloger, W., "A memory allocator," 1996.

[113] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W., "Preventing use-after-free with dangling pointers nullification," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[114] Lee, R. B., Karig, D. K., McGregor, J. P., and Shi, Z., "Enlisting hardware architecture to thwart malicious code injection," in *Security in Pervasive Computing*, pp. 237–252, Springer, 2004.

[115] Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S., "Defeating return-oriented rootkits with return-less kernels," in *European Symposium on Research in Computer Security (ESORICS)*, 2010.

[116] Li, J., Krohn, M. N., Mazières, D., and Shasha, D., "Secure untrusted data repository (sundr)," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[117] Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X., "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures.," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[118] Lin, Z., Riley, R. D., and Xu, D., "Polymorphing software by randomizing data structure layout," in *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2009.

[119] LLVM, "The LLVM compiler infrastructure project." `llvm.org`, 2015.

[120] Lu, K., Song, C., Lee, B., Chung, S. P., Kim, T., and Lee, W., "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[121] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[122] Mao, H., "make sure l2 passes no-alloc acquires through to outer memory." `https://github.com/ucb-bar/uncore/commit/e53b5072caf12a2c18245cecd709204a4231d2d9`, 2015.

[123] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F., "Software fault isolation with api integrity and multi-principal modules," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[124] Mashtizadeh, A. J., Bittau, A., Boneh, D., and Mazières, D., "Ccfi: cryptographically enforced control flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[125] McCalpin, J. D., "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[126] McPhee, W. S., "Operating system integrity in os/vs2," *IBM Systems Journal*, vol. 13, no. 3, pp. 230–252, 1974.

[127] McVoy, L. and Staelin, C., "Lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference (ATC)*, 1996.

[128] Microsoft, "App capability declarations (Windows Runtime apps)." http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx, 2012.

[129] Microsoft Edge Team, "Microsoft edge: Building a safer browser." https://blogs.windows.com/msedgedev/2015/05/11/microsoft-edge-building-a-safer-browser/, 2015.

[130] Min, C., Kashyap, S., Lee, B., Song, C., and Kim, T., "Cross-checking semantic correctness: The case of finding file system bugs," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[131] MITRE, "Cve-2013-6282." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282, 2013.

[132] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B., "Vigilare: Toward snoop-based kernel integrity monitor," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[133] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S., "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[134] Nagarakatte, S., Martin, M. M., and Zdancewic, S., "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[135] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S., "Softbound: highly compatible and complete spatial memory safety for c," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[136] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S., "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management*, 2010.

[137] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W., "Ccured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[138] Necula, G. C. and Lee, P., "Safe kernel extensions without run-time checking," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[139] Nethercote, N. and Seward, J., "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[140] Netzer, R. H. and Miller, B. P., "What are race conditions? some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[141] Newsham, T., "Format string attacks," 2000.

[142] Newsome, J. and Song, D., "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[143] Niu, B. and Tan, G., "Modular control-flow integrity," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[144] Niu, B. and Tan, G., "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[145] Niu, B. and Tan, G., "Per-Input Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[146] Nossum, V., "Getting Started With kmemcheck." `https://www.kernel.org/doc/Documentation/kmemcheck.txt`, 2015.

[147] Novark, G. and Berger, E. D., "DieHarder: Securing the Heap," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[148] Oracle, "Introduction to sparc m7 and application data integrity (adi)." `https://swisdev.oracle.com/_files/What-Is-ADI.html`.

[149] Ou, A., Waterman, A., Nguyen, Q., darius bluespec, and Dabbelt, P., "RISC-V Linux Port." `https://github.com/riscv/riscv-linux`, 2015.

[150] Ozdoganoglu, H., Vijaykumar, T., Brodley, C. E., Kuperman, B., and Jalote, A., "SmashGuard: A hardware solution to prevent security attacks on the function return address," *Computers, IEEE Transactions on*, 2006.

[151] Paolino, M., "ARM TrustZone and KVM Coexistence with RTOS For Automotive," in *ALS Japan*, 2015.

[152] Pappas, V., Polychronakis, M., and Keromytis, A. D., "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[153] Pappas, V., Polychronakis, M., and Keromytis, A. D., "Transparent rop exploit mitigation using indirect branch tracing," in *USENIX Security Symposium (Security)*, 2013.

[154] PATIL, H. and FISCHER, C., "Low-cost, concurrent checking of pointer and array accesses in c programs," *Software: Practice and Experience*, vol. 27, no. 1, pp. 87–110, 1997.

[155] PAX, "Homepage of the pax team." `https://pax.grsecurity.net/`, 2013.

[156] PaX-Team, "PaX Address Space Layout Randomization." `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[157] Payne, B. D., Carbone, M., Sharif, M., and Lee, W., "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[158] Petroni Jr, N. L., Fraser, T., Walters, A., and Arbaugh, W. A., "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data.," in *USENIX Security Symposium (Security)*, 2006.

[159] Petroni Jr, N. L. and Hicks, M., "Automated detection of persistent kernel control-flow attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[160] Pie, P., "Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup," 2013.

[161] Portokalidis, G. and Keromytis, A. D., "Fast and practical instruction-set randomization for commodity systems," in *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[162] project authors, T. V. `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`.

[163] Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y., "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[164] Roglia, G. F., Martignoni, L., Paleari, R., and Bruschi, D., "Surgically returning to randomized lib (c)," in *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[165] Ruwase, O. and Lam, M. S., "A practical dynamic buffer overflow detector," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[166] Schmidt, C., "Low Level Virtual Machine (LLVM)." `https://github.com/riscv/riscv-llvm`, 2014.

[167] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T., "Counterfeit Object-oriented Programming: On the difficulty of preventing code reuse attacks in C++ applications," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[168] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L., "Retargetable and reconfigurable software dynamic translation," in *International Symposium on Code Generation and Optimization (CGO)*, 2003.

[169] Scott, K. and Davidson, J., "Strata: A software dynamic translation infrastructure," tech. rep., University of Virginia, 2001.

[170] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B., "Adapting software fault isolation to contemporary cpu architectures," in *USENIX Security Symposium (Security)*, 2010.

[171] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D., "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference (ATC)*, 2012.

[172] Serna, F. J., "The info leak era on software exploitation," in *Black Hat USA*, 2012.

[173] Seshadri, A., Luk, M., Qu, N., and Perrig, A., "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[174] Seward, J. and Nethercote, N., "Using Valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference (ATC)*, 2005.

[175] Seward, J. and Nethercote, N., "Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference (ATC)*, 2005.

[176] Shacham, H., "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[177] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D., "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[178] Sharif, M. I., Lee, W., Cui, W., and Lanzi, A., "Secure in-vm monitoring using hardware virtualization," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[179] Sintsov, A., "Writing jit-spray shellcode for fun and profit," 2010.

[180] Smalley, S. and Craig, R., "Security enhanced (se) android: Bringing flexible mac to android.," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[181] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R., "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[182] Song, C., Lee, B., Lu, K., Harris, W. R., Kim, T., and Lee, W., "Enforcing kernel security invariants with data flow integrity," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[183] Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., and Paek, Y., "HDFI: Hardware-Assisted Data-Fow Isolation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[184] Song, C., Zhang, C., Wang, T., Lee, W., and Melski, D., "Exploiting and protecting dynamic code generation," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[185] Srivastava, A. and Giffin, J., "Efficient protection of kernel data structures via object partitioning," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[186] Standard Performance Evaluation Corporation, "SPEC CINT2006 Benchmarks." `http://www.spec.org/cpu2006/CINT2006/`.

[187] Standard Performance Evaluation Corporation, "SPEC CPU2000 benchmark descriptions - CINT 2000." `https://www.spec.org/cpu2000/CINT2000/`, 2003.

[188] Stepanov, E. and Serebryany, K., "MemorySanitizer: fast detector of uninitialized memory use in C++," in *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[189] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T., "Breaking the memory secrecy assumption," in *European Workshop on System Security*, 2009.

[190] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S., "Secure program execution via dynamic information flow tracking," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[191] "System error code." `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382%28v=vs.85%29.aspx`, 2001.

[192] Szekeres, L., Payer, M., Wei, T., and Song, D., "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[193] Taylor, I. L., "Linker relro." `http://www.airs.com/blog/archives/189`, 2008.

[194] The IEEE and The Open Group, *errno.h - system error numbers*. The Open Group, 2013. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, `http://pubs.opengroup.org/onlinepubs/9699919799/functions/rename.html`.

[195] The Linux Foundation, "Llvmlinux." `http://llvm.linuxfoundation.org/index.php/Main_Page`.

[196] The MITRE Corporation, "CVE-2013-6282." `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282`.

[197] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G., "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security Symposium (Security)*, 2014.

[198] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P., "On the expressiveness of return-into-libc attacks," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[199] UC Berkeley Architecture Research, "Rocket chip generator." `https://github.com/ucb-bar/rocket-chip`, 2015.

[200] UC Berkeley Architecture Research, "Rocket microarchitectural implementation of RISC-V ISA." `https://github.com/ucb-bar/rocket`, 2015.

[201] Vendicator, "Stack shield: A stack smashing technique protection tool for linux," 2000.

[202] Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M., "Flexitaint: A programmable accelerator for dynamic taint propagation," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[203] Vidas, T., Votipka, D., and Christin, N., "All your droid are belong to us: A survey of current android attacks.," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2011.

[204] Vogl, S., Pfoh, J., Kittel, T., and Eckert, C., "Persistent data-only malware: Function hooks without code," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[205] W3C. http://www.w3.org/TR/workers/, 2012.

[206] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A., "A first step towards automated detection of buffer overrun vulnerabilities.," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2000.

[207] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L., "Efficient software-based fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1994.

[208] Wang, X., Chen, H., Jia, Z., Zeldovich, N., and Kaashoek, M. F., "Improving integer security for systems with kint.," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[209] Wang, Z. and Jiang, X., "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[210] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z., "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[211] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K., "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," Tech. Rep. UCB/EECS-2014-54, UCB, 2014.

[212] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M., "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[213] Wei, T., Wang, T., Duan, L., and Luo, J., "Secure dynamic code generation against spraying," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[214] Weston, D., Miller, M., and Rain, T., "Exploitation trends: From potential risk to actual risk," in *RSA*, 2015.

[215] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W., "RIPE: runtime intrusion prevention evaluator," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[216] Witchel, E., Cates, J., and Asanović, K., "Mondrian memory protection," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[217] Xilinx, "ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC user guide." http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2015.

[218] Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K., "Architecture support for defending against buffer overflow attacks," in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[219] Xu, W., DuVarney, D. C., and Sekar, R., "An efficient and backwards-compatible transformation to ensure memory safety of c programs," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2004.

[220] Yang, J., Kremenek, T., Xie, Y., and Engler, D., "Meca: an extensible, expressive system and language for statically checking security properties," in *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[221] Yang, Z. and Yang, M., "Leakminer: Detect information leakage on android with static taint analysis," in *International Workshop on Computer Science and Engineering (WCSE)*, 2012.

[222] Ye, D., Sui, Y., and Xue, J., "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[223] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N., "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[224] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E., "Panorama: capturing system-wide information flow for malware detection and analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[225] Yong, S. H. and Horwitz, S., "Protecting c programs from attacks via invalid pointer dereferences," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2003.

[226] Younan, Y., "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[227] Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., and Joosen, W., "Paricheck: an efficient pointer arithmetic checker for c programs," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[228] Yu, Y., "Write once, pwn anywhere," in *BlackHat USA*, 2014.

[229] Zeldovich, N., Kannan, H., Dalton, M., , and Kozyrakis, C., "Hardware enforcement of application security policies using tagged memory," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[230] Zhang, C., Carr, S. A., Li, T., Ding, Y., Song, C., Payer, M., and Song, D., "VTrust: Regaining trust on virtual calls," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[231] Zhang, C., Niknami, M., Chen, K. Z., Song, C., Chen, Z., and Song, D., "Jitscope: Protecting web users from control-flow hijacking attacks," in *IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[232] Zhang, C., Song, C., Chen, K. Z., Chen, Z., and Song, D., "VTint: Protecting virtual function tables' integrity," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[233] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W., "Practical control flow integrity and randomization for binary executables," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[234] Zhang, M. and Sekar, R., "Control flow integrity for cots binaries," in *USENIX Security Symposium (Security)*, 2013.

[235] Zhou, Y., Wang, X., Chen, Y., and Wang, Z., "Armlock: Hardware-based fault isolation for arm," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.